Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis in Computer Science
Supervised by
Benoit Baudry and Martin Monperrus

Stockholm, Sweden, March 2024

TRITA-EECS-AVL-2024:10
ISBN 100-

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

WebAssembly, now the fourth officially recognized web language, enables web browsers to port native applications for the Web. Furthermore, WebAssembly has evolved into an essential element for backend scenarios such as cloud computing and edge computing. Therefore, WebAssembly finds use in a plethora of applications, including but not limited to, web browsers, blockchain, and cloud computing. Despite the emphasis on security since its design and specification, WebAssembly remains susceptible to various forms of attacks, including memory corruption and side-channels. Furthermore, WebAssembly has been manipulated to disseminate malware, particularly in cases of browser cryptojacking.

Web page resources, including those containing WebAssembly binaries, are predominantly served from centralized data centers in the modern digital landscape. In conjunction with browser clients, thousands of edge devices operate millions of identical WebAssembly instantiations every second. This phenomenon creates a highly predictable ecosystem, wherein potential attackers can anticipate behavior either in browsers or backend nodes. Such predictability escalates the potential impact of vulnerabilities within these ecosystems, paving the way for high-impact side-channel and memory attacks. For instance, a flaw in a web browser, triggered by a defective WebAssembly program, holds the potential to affect millions of users.

This work aims to harden the security within the WebAssembly ecosystem through the introduction of Software Diversification methods and tools. Software Diversification is a strategy designed to augment the costs of exploiting vulnerabilities by making software less predictable. The predictability within ecosystems can be diminished by automatically generating different, yet functionally equivalent, program variants. These variants strengthen observable properties that are typically used to launch attacks, and in many instances, can completely eliminate such vulnerabilities.

This work introduces three tools: CROW, MEWE, and WASM-MUTATE. Each tool has been specifically designed to tackle a unique facet of Software Diversification. We present empirical evidence demonstrating the potential application of our Software Diversification methods to WebAssembly programs in two distinct ways: Offensive and Defensive Software Diversification. Our research into Offensive Software Diversification in WebAssembly unveils potential paths for enhancing the detection of WebAssembly malware. On the other hand, our experiments in Defensive Software Diversification show that WebAssembly programs can be hardened against side-channel attacks, specifically the Spectre attack.

Keywords: WebAssembly, Software Diversification, Side-Channels

Sammanfattning

WebAssembly, nu det fjärde officiellt erkända webbspråket, gör det möjligt för webbläsare att portera nativa applikationer till webben. Dessutom har WebAssembly utvecklats till en väsentlig komponent för backend-scenarier såsom molntjänster och edge-tjänster. Därmed används WebAssembly i en mängd olika applikationer, däribland webbläsare, blockchain och molntjänster. Trots sitt fokus på säkerhet från dess design till dess specifikation är WebAssembly fortfarande mottagligt för olika former av attacker, såsom minneskorruption och sidokanalattacker. Dessutom har WebAssembly manipulerats för att sprida skadlig programvara, särskilt otillåten cryptobrytning i webbläsare.

Webbsideresurser, inklusive de som innehåller exekverbar WebAssembly, skickas i en modern digital kontext huvudsakligen från centraliserade datacenter. Tusentals edge-enheter, i samarbete med webbläsarklienter, kör miljontals identiska WebAssembly-instantieringar varje sekund. Detta fenomen skapar ett högst förutsägbart ekosystem, där potentiella angripare kan förutse beteenden antingen i webbläsare eller backend-noder. En sådan förutsägbarhet ökar potentialen för sårbarheter inom dessa ekosystem och öppnar dörren för sidkanal- och minnesattacker med stor påverkan. Till exempel kan en brist i en webbläsare, framkallad av ett defekt WebAssembly-program, ha potential att påverka miljontals användare.

Denna avhandling syftar till att stärka säkerheten inom WebAssembly-ekosystemet genom införandet av metoder och verktyg för mjukvarudiversifiering. Mjukvarudiversifiering är en strategi som är utformad för att öka kostnaderna för att exploatera sårbarheter genom att göra programvaran oförutsägbar. Förutsägbarheten inom ekosystem kan minskas genom att automatiskt generera olika programvaruvarianter. Dessa varianter förstärker observerbara egenskaper som vanligtvis används för att starta attacker och kan i många fall helt eliminera sådana sårbarheter.

Detta arbete introducerar tre verktyg: CROW, MEWE och WASM-MUTATE. Varje verktyg har utformats specifikt för att hantera en unik aspekt av mjukvarudiversifiering. Vi presenterar empiriska bevis som visar på potentialen för tillämpning av våra metoder för mjukvarudiversifiering av WebAssembly-program på två distinkta sätt: offensiv och defensiv mjukvarudiversifiering. Vår forskning om offensiv mjukvarudiversifiering i WebAssembly avslöjar potentiella vägar för att förbättra upptäckten av WebAssembly-malware. Å andra sidan visar våra experiment inom defensiv mjukvarudiversifiering att WebAssembly-program kan härdas mot sidokanalattacker, särskilt Spectre-attacken.

LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023, 17 pages
   https://www.sciencedirect.com/science/article/pii/S0167404823002067

2. ***WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
   Submitted to Computers & Security, under revision, 20 pages
   https://arxiv.org/pdf/2309.07638.pdf

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Workshop on Moving Target Defense (MTD 2022), 12 pages
   https://dl.acm.org/doi/abs/10.1145/3560828.3564007

4. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
   https://doi.org/10.14722/madweb.2021.23004

5. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs, 4 pages
   https://doi.org/10.1145/3397537.3397567

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019), 10 pages
   https://doi.org/10.1145/3358504.3361228

ACKNOWLEDGEMENT

Contents

Part I

Thesis

1 INTRODUCTION

Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [1]*

THE first web browser, Nexus, made its appearance in 1990 [2]. At its inception, web browsing consisted solely of retrieving and displaying small, static text pages. With Nexus, users could access for the first time interlinked hypertext documents, so-called HTML pages. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet-based companies, and the demand for more engaging user experiences gave rise to the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [3], a programming language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers offer JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Consequently, over the past several decades, web browsers have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers.

Despite being the main scripting language in modern web browsers, JavaScript has limitations due to its unique language characteristics [4]. Each JavaScript engine requires the parsing and recompiling of the JavaScript code, thereby causing substantial overhead. In practice, the process of parsing and compiling JavaScript code constitutes the majority of website load times[1]. Additionally, JavaScript presents security issues, including the lack of memory isolation, which potentially enables information extraction from other processes [5, 6]. Numerous

[1] `https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast`

attempts have been made to port other languages, offering different guarantees, to the browser execution as alternatives to JavaScript. For instance, Java applets emerged on web pages in the late 90s, enabling the execution of Java bytecode on the client side[2]. Likewise, Microsoft proposed ActiveX in 1996[3], and Silverlight in 2007[4]. However, these attempts either failed to persist or experienced low adoption, primarily due to security issues and the absence of consensus among the community of browser vendors.

1.1 WebAssembly

Importantly, in 2014, Alon Zakai et al. proposed the Emscripten tool[5]. Emscripten employs a strict subset of JavaScript, asm.js, to facilitate the compilation of low-level code such as C to JavaScript. Asm.js was included as an LLVM backend[6]. This strategy offered the advantages of the ahead-of-time optimizations from LLVM, resulting in performance gains on browser clients[7] when compared to standard JavaScript code. Asm.js outperformed JavaScript because it restricted language features to those that could be optimized in the LLVM pipeline. Moreover, it eliminated most of the language's dynamic characteristics, limiting it to numerical types, top-level functions, and one large array in memory accessed directly as raw data. Asm.js proved that client-side code could be enhanced with the appropriate language design and standardization. In response to persistent JavaScript-related issues, the formalization and creation of a formal specification following asm.js laid the groundwork for the emergence of WebAssembly as a fast, low-level, portable bytecode for browsers. In 2015, the Web Consortium (W3C) standardized WebAssembly. As a result, WebAssembly bytecode became the fourth official language for the web.

The first distinction from earlier attempts to port non-JavaScript languages to the web lies in WebAssembly's initial design. Unlike its predecessors, WebAssembly was crafted to supplement JavaScript in the browser as a platform-agnostic, low-level bytecode, rather than to completely replace it. Its primary goal was to replace computing-intensive JavaScript code in contemporary web applications. Additionally, WebAssembly is the inaugural major language that used formal specification and verification right from the design inception [7, 8].

Importantly, WebAssembly provides a platform for compiling several legacy code applications, like those written in C/C++. For example, LLVM includes

[2] https://www.oracle.com/java/technologies/javase/9-deprecated-features.html

[3] https://web.archive.org/web/20090828024117/http://www.microsoft.com/presspass/press/1996/mar96/activxpr.mspx

[4] https://www.microsoft.com/silverlight/

[5] https://emscripten.org/

[6] http://asmjs.org/spec/latest/

[7] https://hacks.mozilla.org/2015/03/asm-speedups-everywhere/

WebAssembly as a backend since release 7.1.0 published in May 2019[8]. WebAssembly paves the way for web applications to undertake roles traditionally reserved for native desktop applications, having the potential to transform web software as we know it. For example, applications such as AutoCAD and Adobe Photoshop have been ported to WebAssembly[9].

The WebAssembly specification embodies several language design principles that pave the way for its extension beyond the web ecosystem. For instance, the architecture of WebAssembly guarantees self-containment. Inherently, WebAssembly binaries are prohibited from accessing memory beyond their own designated space, thereby amplifying security via Software Fault Isolation (SFI) policies [9]. Consequently, research has highlighted the benefits of integrating WebAssembly as an intermediate layer in contemporary cloud platforms [10]. In particular, the employment of WebAssembly binaries improves startup times and optimizes memory consumption, outperforming virtualization and containerization [11]. Furthermore, compared to virtual machines and containers, WebAssembly programs are more compact, highlighting their efficient deployment, especially when network transportation is a consideration. The methodology for standalone WebAssembly execution was formalized in 2019 when the Bytecode Alliance proposed the WebAssembly System Interface (WASI)[10]. WASI standardizes the execution of WebAssembly via a POSIX-like interface protocol, thereby facilitating the execution of WebAssembly closer to operating systems. This standardization enables WebAssembly to function outside web browsers, extending its use to cloud environments and IoT devices [12, 13].

The extensive applicability and rapid adoption of WebAssembly have prompted requests for additional features. However, these demands do not always align with the initial specifications. For extending WebAssembly with a new proposal, it must satisfy particular criteria. A new proposal needs a formal specification and, at least two implementations, e.g., two different WebAssembly engines. This approach allows for swift incorporation of new formalization and features via the so-called "evergreen method" while maintaining the original WebAssembly specification intact. Since the inception of WebAssembly, numerous extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. After approval, new extensions remain optional, ensuring that the core WebAssembly version remains 1.0. The ongoing development of WebAssembly provides avenues for research and development. However, it also gives rise to security concerns within the ecosystem, as new threats emerge.

[8] https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0

[9] https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw

[10] https://github.com/WebAssembly/WASI

1.2 Predictability in WebAssembly ecosystems

Over the past three decades, web browsers and JavaScript have had significant evolution, leading to a myriad of implementations. However, only Firefox, Chrome, Safari, and Edge are typically used on devices. Web page resources, including those containing WebAssembly binaries, are primarily served from centralized datacenters [14]. This situation creates a highly predictable ecosystem, where potential attackers can predict ecosystem behavior, from the browser to the code it executes. This predictability may be exploited to launch large-scale attacks, as predictability inherently increases the chances of successful attacks [15]. For example, if one-quarter of all devices operate the same code in the same browser, a single flaw could impact millions of devices in the same way [16].

The aforementioned issue is exacerbated when considering the adoption of WebAssembly by edge-cloud computing platforms to provide services. In addition to browser clients, thousands of edge devices operate millions of identical WebAssembly instantiations per second [17]. This suggests that a single vulnerable WebAssembly binary in an edge network node could render every node identically susceptible due to the binary replication occurring on each node. A potential attacker could compromise all edge nodes concurrently, implying that a single distributed WebAssembly binary could trigger a global attack[11].

We devise two scenarios where predictability affects WebAssembly ecosystems. First, the predictability of execution engines and WebAssembly binaries themselves facilitates side-channel and memory attacks. Despite the praise for WebAssembly's security, particularly its design that prohibits programs from accessing data beyond their own memory, it is not immune to such vulnerabilities. For example, Rokicki et al. highlighted the potential risk of port contention side-channel attacks using WebAssembly malware in browsers [18]. In such cases, mitigations often involve hardware and operating-level changes, which are not always feasible. Moreover, attacks within the memory of WebAssembly itself are feasible [19, 20] as innate vulnerabilities can exist in WebAssembly binaries due to flaws in the source code. Besides, the lack of stack-smashing protections could result in unnoticed overflows and crashes during WebAssembly executions [21]. In standalone deployments, Genkin et al. demonstrated the possibility of data extraction via cache-timing side channels in WebAssembly [22]. In a similar vein, Maisuradze and Rossow exhibited speculative execution attacks on WebAssembly binaries [23].

Second, the defenses for identifying and addressing vulnerabilities are generally predictable. In particular, this predictability can be manipulated by malicious actors to create programs aimed at deceiving these defense mechanisms. For example, malware can be distributed via WebAssembly binaries. The

[11] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

capability of WebAssembly for efficient computation makes it an appealing target for misuse by cybercriminals, especially for cryptojacking [24]. The challenge in identifying and eliminating cryptojacking enables it to function persistently on a victim's computer, constantly utilizing resources and generating income for the attacker [25]. Several techniques, such as static analysis, dynamic analysis, and even sophisticated machine learning methods, are successfully applied to detect WebAssembly malware [26, 27, 28, 29, 30, 31]. However, most of these research works do not consider the predictability of an attacker knowing that a WebAssembly program is not treated as obfuscated.

1.3 Problem statements

To sum up, predictability and potential vulnerabilities form a harmful combination. This principle does not exclude WebAssembly and its ecosystem. The effect of exploiting a single vulnerability in WebAssembly could prove catastrophic, given all devices running the same WebAssembly binaries could be affected. On the other hand, WebAssembly malware pose a severe threat. Present defenses may not adequately protect against them, as they have not been designed to manage situations outside predictable scenarios, such as obfuscation. Besides, mitigations might require hardware and operating-level changes, which are not always feasible. In this dissertation, we tackle the subsequent two problems:

P1 The WebAssembly ecosystem and binaries are susceptible to attacks, especially those from side-channel threats.

P2 WebAssembly malware presents a substantial threat. Predictability leads to the assumption that malware is typically considered unique.

1.4 Approach: Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problems via Software Diversification. Software Diversification is a security strategy that involves identifying, developing, and deploying program variants of a given original program [32]. Pioneers in this field, Cohen et al. [33] and Forrest et al. [34], proposed enhancing software diversity through code transformations. Their proposal recommended the creation of diverse program variants, maintaining their original functionalities. Software Diversification aims to lessen potential vulnerabilities by enhancing behavior unpredictability in observabilities used to conduct attacks, e.g., side-channels.

Eichin et al. underscored the practical benefits of diversification [35] early in 1989. They illustrated how diversification limited the exploitation of the Morris Worm to a few machines. From an attacker's perspective, the diversity of target systems rendered them unpredictable. Therefore, Software Diversification effectively removes vulnerabilities. For WebAssembly, Software Diversification

Contribution	Research papers			
	I [36]	II [37]	III [38]	IV [39]
C1 Offensive diversification technique				✓
C2 Defensive diversification technique	✓	✓	✓	
C3 Extensive experimental evaluation	✓	✓	✓	✓

Table 1.1: *Mapping between contributions and research papers.*

could bolster browsers and standalone engines by providing diversified program variants, making it harder for attackers to exploit vulnerabilities, addressing **P1**. Furthermore, it could increase the accuracy of WebAssembly malware detectors and WebAssembly analysis tools in general, addressing **P2**. However, the implementation of Software Diversification in WebAssembly is still largely unexplored. In light of this, we offer the following contributions within the context of Software Diversification, which are not necessarily mutually exclusive.

C1 Offensive Diversification Technique: In order to address **P2**, we evaluate the potential for using generated WebAssembly program variants for offensive purposes. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

C2 Defensive Diversification Technique: In order to address **P1**, we assess how diversified WebAssembly program variants could be used for defensive purposes. We provide empirical insights about the practical usage of the generated variants in preventing attacks.

C3 Extensive experimental evaluation: For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research on **P1** and **P2**.

1.5 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

I: **CROW: Code randomization for WebAssembly bytecode.**
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry,

Martin Monperrus
Workshop on Measurements Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

Summary: In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements Software Diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create thousands of statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

II: **Multivariant execution at the Edge.**
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Workshop on Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

Summary: In this paper, we synthesize functionally equivalent variants of deployed edge services. Service variants are encapsulated into a single multivariant WebAssembly binary. A random variant is selected and executed each time a function is invoked. Execution of multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

III: **WASM-MUTATE: Fast and efficient Software Diversification for WebAssembly.**
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Submitted to Computers & Security, under revision, 20 pages
`https://arxiv.org/pdf/2309.07638.pdf`

Summary: This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly generate functionally equivalent yet behaviorally diverse WebAssembly variants by randomly traversing e-graphs. E-graphs are specific graph data structures for representing and applying rewriting rules. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in minutes. Importantly, WASM-MUTATE can

safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

IV: **WebAssembly Diversification for Malware evasion.**
Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages

Summary: WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiming WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

1.6 Thesis outline

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [40]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.

2 BACKGROUND AND STATE OF THE ART

You must have a map, no matter how rough. Otherwise you wander all over the place.

— J.R.R. Tolkien

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss WebAssembly, focusing on its design and security model. Besides, we discuss the current state-of-the-art of WebAssembly research. In Section 2.2 we discuss related works in the area of Software Diversification. Moreover, we delve into the open challenges regarding the diversification of WebAssembly programs.

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported by all major web browser vendors. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al [7]. It is designed to be fast, portable, self-contained, and secure.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Previous works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup time and memory usage than containerization and virtualization [10, 11]. Consequently, in 2019, the Bytecode Alliance proposed WebAssembly System Interface (WASI) [41]. WASI pioneered the execution of WebAssembly with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of WebAssembly in heterogeneous platforms [42], i.e., IoT and Edge computing [43, 44].

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [45]. Other applications include text processing, visualization, media processing, programming language testing, online gambling, bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly

suitable for Function-as-a-Service (FaaS) platforms [12] like Cloudflare and Fastly. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed also apply to future WebAssembly versions.

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go ahead of time, which means that Wasm binaries can benefit from the optimizations of the source language compiler. The resulting WebAssembly program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

In Listing 2.1 and Listing 2.2, we present a Rust program alongside its corresponding WebAssembly binary. The Rust program in Listing 2.1 iteratively calculates the Fibonacci sequence up to a given number that comes from the host engine. The code in the program encompasses various elements such as vector allocations, external function usage, and a function definition that includes a loop, conditionals, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is simplified in its textual format, known as WAT[1]. The function prototype in lines 4 and 5 of Listing 2.1 are converted into an imported function, as seen in lines 8 and 9 of Listing 2.2. The `fibonacci` function, spanning lines 7 to 20 in Listing 2.1, is compiled into a Wasm function covering lines 14 to 31 in Listing 2.2. Within this function, the translation of various Rust language constructs into Wasm can be observed. For instance, the `for` loop found in line 14 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The breaking condition of the loop is transformed into a conditional branch, as depicted in line 23 of Listing 2.2. In this scenario, the function yields the final set value in the `local` variable. Note that for optimization purposes, the loop concludes by returning the result value, instead of returning post-completion of the loop.

[1]The WAT text format is primarily designed for human readability and for low-level manual editing.

```
...
// Imported form host
extern "C" {
    fn log(s: &str);
    fn get_input() -> usize; }

fn fibonacci(n: usize) -> i32 {
    // Iterative fibonacci
    // Create a vector of size n+1
    let mut fibo_result = vec![0; n + 1];
    // Set ith 0 and 1
    fibo_result[0] = 1;
    fibo_result[1] = 1;
    for i in 2..=n {
        // f[i] = f[i-1] + f[i-2]
        fibo_result[i] = fibo_result[i - 1] + fibo_result[i - 2];
    }
    // Return the last element
    return fibo_result[n];
}
// Pub to export the function
pub fn main() {
    // Get the input from the user
    let ith = get_input();
    // Calculate the fibonacci
    let fib = fibonacci(get_input());
    // Print the result in the host imported function
    log(&format!("{}",fib));
}
```

Listing 2.1: *Example Rust program which includes external function usage, a function definition featuring a loop, function calls, imported functions, and memory accesses.*

There are several compilers that turn source code into WebAssembly binaries. For example, LLVM compiles to WebAssembly as a backend option since its 7.1.0 release in early 2019[2], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig [45] reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a modular and state-of-the-art optimization infrastructure for WebAssembly binaries. Today, Emscripten[4] is the most frequently used tool for porting C/C++ code to the Web as a drop-in replacement for a standard compiler like gcc or clang. The main advantage of Emscripten is that it provides a complete toolchain for compiling C/C++ code to WebAssembly, including the automatic generation of the external functions for interacting with a Web host environment. Recently, the Kotlin Multiplatform framework[5] has incorporated WebAssembly as a compilation target, enabling the compilation of Kotlin code to WebAssembly. Similarly, the Cheerp[6] project proposes a Java Virtual Machine(JVM) fully ported to WebAssembly, supporting Java applications and legacy applets in the browser.

[2] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[3] A subset of the TypeScript language
[4] `https://emscripten.org/docs/tools_reference/emcc.html`
[5] `https://kotlinlang.org/docs/wasm-overview.html`
[6] `https://labs.leaningtech.com/blog/cheerpj-3-deep-dive`

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
  ...
  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
  (type (;type index 0;) (func (param i32 i32)))
  ...
  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
  (import "__wbg__" "__wbg_log" (func (;1;) (type 0)))
  (import "__wbg__" "__wbg_getinput" (func (;2;) (type 8)))
  ...
  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
  (@custom "name" "...")
  ...
  (func (;func index 40;) (type 1) (param i32) (result i32)
    (local i32 i32 i32 i32 i32) ;local variables;
    ...
    loop  ; label = @1 ;
      ...
      i32.eqz
      if  ; label = @2 Compare the top of the stack ;
        ...
        local.get 0
        return ; Return the last element which is saved in local 0 ;
      end
      ...
      block  ;label = @2 ;
        ...
        i32.store ; Store the fib value in the mem assigned to the
            ↪ result array;
        br 1 (;@1;) ;Continue the loop;
      end
    end)
  ...
  (func (;44;) (type 8) (result i32)
    ...
    call 2 ; Calling the imported function to get input ;
    i32.store ; Store the input in memory ;
  ...
  (func (;45;) (type 7)
    (local i32 i32 i32)
    ...
    call 44
    call 40 ; Calling fibo function ;
    i32.store offset=20
  ...
  (table (;0;) 33 33 funcref)
  ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
  (memory (;0;) 17)
  ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
  (global (;global index 0;) (mut i32 ;mut global;) (i32.const 1048576))
  ...
  ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
  (export "memory" (memory 0))
  (export "fibo" (func 40))
  (export "main" (func 45))
  ...
  ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
  (data (;data segment index 0;) (i32.const 1048576) "invalid args...")
  ...
; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
  (@custom "producers" "...")
```

Listing 2.2: *Refer to Listing 2.1 for the Rust code example. This example showcases the translation from Rust to Wasm. For clarity, we have marked elements and portions of the WebAssembly binary as comments.*

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[7] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[8] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, packaging the interpreter and the code in one single standalone WebAssembly binary is still immature and faces challenges. For example, the absence of JIT compilation for the "interpreted" code makes it less suitable for long-running tasks [46, 47]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

2.1.2 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A WebAssembly binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty. For clarity, we also annotate sections as comments in the Wasm code in Listing 2.2.

A WebAssembly binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to boost the compilation process through parallel parsing. Moreover, the *Code Section*'s instructions are further compacted through the use of the LEB128[9] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

[7] `https://github.com/bytecodealliance/javy`
[8] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`
[9] `https://en.wikipedia.org/wiki/LEB128`

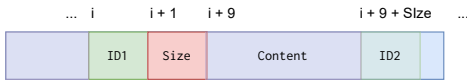


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

2.1.3 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of WebAssembly programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control-flow and the analysis of WebAssembly binaries.

Execution Stack: At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Section 2.1.2. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instruction starts and loop starts. Control frames manage function calls and function returns. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

Functions: At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, reads from the linear memory, writes to the linear memory, reads a global, writes a global or modifies the control-flow of the function. Recalling the example WebAssembly binary, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 15 and Line 19 in Listing 2.2. When an instruction reads its operands from the stack, it pushes back the result. Notice that, numeric instructions are annotated with their corresponding type.

Memory model: A WebAssembly module instance incorporates three key types of memory-related components: linear memory, local variables and global variables. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [20]. Managed refers to components that are exclusively modified by the host engine at the lowest level,

e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include a linear memory instance, which is a contiguous array of bytes. This linear memory is accessed using 32-bit integers (`i32`) and is shareable only between the initiating engine and the WebAssembly module instance. Generally, the linear memory is considered to be unmanaged, e.g., line 28 of Listing 2.2 shows an explicit memory access opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 49 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance (e.g., line 15 and line 22 in Listing 2.2). They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

WebAssembly module execution: While a WebAssembly binary could be interpreted, the most practical approach is to JIT compile it into machine code [48]. The main reason is that WebAssembly is optimized and closely aligned with machine code, leading to swift JIT compilation for execution. Browser engines such as V8[10] and SpiderMonkey[11] use this strategy when executing WebAssembly binaries in browser clients. In practice, browsers initially employ a baseline compiler to ensure the rapid availability of incoming WebAssembly binaries. Simultaneously, an optimizing compiler operates in the background. Consequently, the first generated machine code is eventually supplanted by the optimized version. Once JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment exclusively through imported functions. This sandboxing follows the Software Fault Isolation (SFI) guarantee, meaning that a WebAssembly program cannot arbitrarily access code or data of its runtime.

WebAssembly standalone engines: While initially intended for browsers, WebAssembly has undergone significant evolution, primarily due to WASI [41]. WASI establishes a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate WebAssembly binaries that implement WASI, which allows execution in standalone engines. These binaries can then be executed by standalone engines across a variety of environments, including the cloud, servers, and IoT devices [49, 48]. Similarly to browsers, these engines often translate WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process.

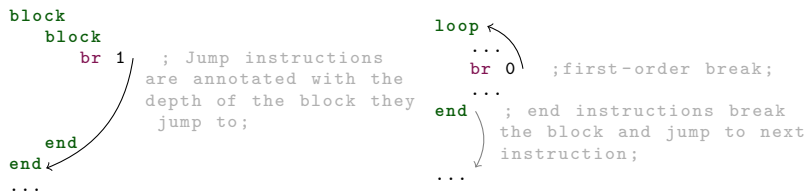
[10] `https://chromium.googlesource.com/v8/v8.git`
[11] `https://spidermonkey.dev/`

Standalone engines such as WASM3[12], Wasmer[13], Wasmtime[14], WAVM[15], and Sledge [50] have been developed to support both WebAssembly and WASI.

2.1.4 WebAssembly's control-flow

A WebAssembly function groups instructions into blocks, with the function's entrypoint acting as the root block. In contrast to conventional assembly code, control-flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they start and end. By design, a block cannot reference or execute code from external blocks.

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 31 of Listing 2.2, the execution would proceed to line 32. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 29 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the leftmost part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design

[12] `https://github.com/wasm3/wasm3`
[13] `https://wasmer.io/`
[14] `https://github.com/bytecodealliance/wasmtime`
[15] `https://github.com/WAVM/WAVM`

hardens the rewriting of WebAssembly binaries. For instance, if an outer block is removed, the depth of the break instructions within nested blocks must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control-flow graph to determine the enclosing block depth for each break instruction.

Notice that, WebAssembly's control-flow design adheres to a Control Flow Integrity (CFI) policy. CFI is a security mechanism that limits a program's control-flow to a specified set of valid targets, thereby preventing arbitrary jumps [51]. Thus, even when a WebAssembly program originates from potentially untrustworthy sources, CFI policy theoretically guarantees the prevention of arbitrary jumps to random code locations.

2.1.5 Security and reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need.This section reviews the most relevant works in this field. We group them by the technique they employ.

Static analysis: SecWasm [52] uses information control-flow checking to identify secrecy leaking in WebAssembly binaries. Similarly, Wasmati [53] employs code property graphs for this purpose. Wasp [54] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. CT-Wasm [55], verifies the constant time implementation of cryptographic algorithms in WebAssembly. Similarly, Vivienne applies relational symbolic execution to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations [56]. While these tools emphasize specific strategies, others adopt a more holistic approach. For example, both Wassail [57] and WasmA [58] provide a comprehensive static analysis framework for WebAssembly binaries.

Dynamic analysis: Dynamic analysis involves tools such as TaintAssembly [59], which conducts taint analysis on WebAssembly binaries. Furthermore, Stiévenart et al. have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[60, 61]. This technique aids in debugging, understanding programs, and conducting security analysis. However, Wasabi [62] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. These tools typically analyze software behavior during execution, making them inherently reactive.

Protecting WebAssembly binaries and runtimes: The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and therefore protecting the security of the

entire execution ecosystem. For instance, Swivel [63] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries in Function-as-a-Service (FaaS) platforms by linearizing the machine code from compiling a WebAssembly binary. Similarly, Kolosick et al. [64] modify the Lucet compiler to use zero-cost transitions, eliminating the performance overhead of SFI guarantees implementation. In addition, WaVe [65] introduces a mechanized engine for WebAssembly that facilitates differential testing. WaVe can be employed to detect anomalies in engine implementations running Wasm-WASI programs.

WebAssembly malware: Since the introduction of WebAssembly, the Web has consistently experienced an increase in cryptomalware. This rise primarily stems from the shift of mining algorithms from CPUs to WebAssembly, a transition driven by notable performance benefits [66]. Tools such as MineSweeper [26], MinerRay [27], and MINOS [28] employ static analysis with machine learning techniques to detect browser-based cryptomalware. In addition, SEISMIC [29], RAPID [30], and OUTGuard [31] leverage dynamic analysis techniques to achieve a similar objective. Moreover, VirusTotal[16], a tool incorporating over 60 commercial antivirus systems as black-boxes, is capable of detecting cryptomalware in WebAssembly binaries. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bhansali et al.'s seminal work [67] demonstrates that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

2.1.6 Open challenges

Despite progress in WebAssembly analysis, numerous challenges remain. WebAssembly, though deterministic and well-typed by design, is susceptible to a variety of security threats. First, most existing WebAssembly research is reactive, focusing on detecting and fixing vulnerabilities already reported. This approach leaves WebAssembly binaries and runtime implementations potentially open to unidentified attacks. Second, side-channel attacks present a significant risk. Genkin et al., for example, illustrated how WebAssembly could be manipulated to extract data via cache timing-side channels [22]. Furthermore, research conducted by Maisuradze and Rossow demonstrated the potential for speculative execution attacks on WebAssembly binaries [23]. Rokicki et al. disclosed the possibility for port contention side-channel attacks on WebAssembly binaries in browsers [18]. Finally, the binaries themselves may be inherently vulnerable. For example, studies by Lehmann et al. and Stiévenart et al. suggested that flaws in C/C++ source code could infiltrate WebAssembly binaries [20, 21].

[16] `https://www.virustotal.com`

2.2 Software diversification

Software diversification involves the synthesis, reuse, distribution, and execution of different, functionally equivalent programs. Along with this work, we refer to those programs as software variants. As outlined in Baudry et al.'s survey [68], Software Diversification falls into five usage categories: reusability [69],performance [70], fault tolerance [71], software testing [72], and security [33]. Our work specifically contributes to the last two categories. Based on the works of Cohen et al. [33], Forrest et al. [34], Jackson et al. [73] and Baudry et al. [68], this section presents core concepts and related works. Notice that, we do not regard program variants from Software Product Lines [74] as instances of software diversification. The primary reason is that, by design, Software Product Lines do not produce functionally equivalent programs.

Software variants in our context refer to functionally equivalent versions of an original program, produced through Software Diversification at various stages of the software lifecycle, from dependencies (coarse-grained) to machine code levels (fine-grained). The main goal of Software Diversification is to increase the cost of exploitation by making software less predictable. Diversification may be natural [68] or automatic [75]. Natural diversity refers to the side effect of humans creating software variants using different programming languages, compilers, and operating systems [68], all of which adhere to the initial requirements. The software market and competition typically address the creation of natural diversity. For example, Firefox and Chrome web browsers demonstrate natural diversity due to their practical differences in implementation and performance, despite serving the same purpose. This logic extends to operating systems, database engines, virtual machines, and application servers [68]. Natural diversity significantly aids in system security, as different variants are not susceptible to the same vulnerabilities [76, 77]. Unlike N-Version programming [78], natural diversity organically emerges over decades. In other words, while it does not require the allocation of additional human effort, natural diversity cannot be automatically generated. This is because it is a side effect of the software development process. Given that WebAssembly is a relatively new technology, natural diversity is presently not a feasible option. Hence, for WebAssembly, feasible options are systematic and automatic diversification approaches.

2.2.1 Automatic generation of software variants

The concept of automatic software variants starts with Randell's 1975 work [79], which put forth the notion of artificial fault-tolerant instruction blocks. Artificial Software Diversification, as proposed by Cohen and Forrest in the 1990s [33, 34], gets its development through rewriting strategies. These strategies consist of sets of rewriting rules for modifying software components to create functionally equivalent, yet distinct, programs. Rewriting strategies typically take the form of tuples: `instr1 => (instr2, instr3, ...)`, where `instr` represents the

original code and `(instr2, instr3, ...)` denotes the functionally equivalent code.

Rewriting strategy: The automatic creation of Software Diversification begins with creating rewriting rules. A rewriting rule refers to a functionally equivalent substitution for a code segment, manually written. These rules can be applied at varying levels, from coarse to fine-grained. This can range from the program dependencies level [80] to the instruction level [75]. For example, Cleemput et al. [81] and Homescu et al. [82] inject NOP instructions to yield statically varied versions at the instruction level. Here, the rewriting rule is represented as `instr => (nop instr)`, signifying an insertion of a `nop` operation preceding the instruction.

Instruction Reordering: This strategy reorders instructions in a program. For example, variable declarations may change if compilers reorder them in the symbol tables. This prevents static examination and analysis of parameters and alters memory locations. In this area, Bhatkar et al. [83, 84] proposed the random permutation of variable and routine order for ELF binaries. Such strategies are not implemented for WebAssembly to the best of our knowledge.

Adding, Changing, Removing Jumps and Calls: This strategy generates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [33] primarily illustrated this concept by inserting random jumps in programs. Pettis and Hansen [85] suggested splitting basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [86] de-inlined basic blocks of code as an LLVM pass. In their approach, each de-inlined code transforms into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [84] extended their previous approach [83], replacing function calls with indirect pointer calls in C source code, allowing post-binary reordering of function calls. In the WebAssembly context, thesimilar work is Wobfuscator [87]. Wobfuscator, a JavaScript obfuscator, substitutes pieces of JavaScript code with WebAssembly code, e.g., numeric calculi. This strategy effectively uses the interleaving of calls between JavaScript and WebAssembly to provide JavaScript variants.

Program Memory and Stack Randomization: This strategy alters the layout of programs in the host memory. Additionally, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [83, 84] proposes to randomize the base addresses of applications and library memory regions in ELF binaries. Tadesse Aga and Autin [88], and Lee et al. [89] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [90] suggest separating a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [91] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization. This makes it very

challenging for an attacker to ignore the key to inject executable code. This strategy disrupts the predictability of program execution and mitigates certain exploits such as speculative execution. No work has been found that explicitly applies this strategy to WebAssembly.

ISA Randomization and Simulation: This strategy involves using a key to cipher the original program binary into another encoded binary. Once encoded, the program can only be decoded at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [92], and Barrantes et al. [93] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [94] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [95] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Generally, *ISA randomization and simulation* usually faces a performance penalty, especially for WebAssembly, due to the decoding process as shown in WASMixer evaluation [96].

Code obfuscation: Code obfuscation can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client [97]. Yet, both strategies aim to tackle static reverse engineering of programs. In the context of WebAssembly, Romano et al. [87] proposed an obfuscation technique, wobfuscator, for JavaScript in which part of the code is replaced by calls to complementary WebAssembly functions. Yet, wobfuscator targets JavaScript code, not WebAssembly binaries.

Enumerative synthesis: Enumerative synthesis is a fully automated and systematic approach to generate program variants. It examines all possible programs specific to a given language. The process of enumerative synthesis commences with a piece of input program, typically a basic block. Incrementally, using a defined grammar, it generates all programs of size n. A generated program is then checked for equivalence to the original program, either by using a test suite or a theorem solver. If the generated variant is proven equivalent, it is added to the variant's collection. The procedure continues until all potential programs have been explored. This approach proves especially effective when the solution space is relatively small or can be navigated efficiently. Jacob et al. [98] implemented this strategy for x86 programs. They named this technique superdiversification, drawing parallels to superoptimization [99]. Since this strategy fully explores a program's solution space, it contains the aforementioned strategies as special cases. The application of enumerative synthesis to WebAssembly has not been explored.

2.2.2 Equivalence Checking

Equivalence checking between program variants is a vital component for any program transformation task, ranging from checking compiler optimizations [100] to the artificial synthesis of programs discussed in this chapter. It proves that two pieces of code or programs are functionally equivalent [101]. We can roughly simplify the checking process with the following property: two programs are deemed equivalent if they generate identical outputs when given identical inputs from a closed collection of inputs [102]. We adopt this definition of *functional equivalence modulo input* throughout this dissertation. In Software Diversification, equivalence checking seeks to preserve the original functionality of programs while varying observable behaviors. Two programs, for instance, can differ statically and still compute the same result. We outline three methods to check variant equivalence: by construction, check modulo tests and proof-driven equivalence checking.

Equivalence by construction: The equivalence property can be guaranteed by construction. As previously mentioned, Cleemput et al. [81] and Homescu et al. [82] exemplify transformation strategies that generate semantically equivalent program variants. These variants are equivalent by construction. In their case, NOP instructions produce statically different variants. NOP operations, interleaved by any other type of original instruction, serve as a functionally equivalent replacement. However, developer errors may occur during this process, necessitating further validation. The test suite of the original program can serve as a check for the variant.

Checking modulo tests: The process of checking modulo tests involves utilizing a test suite to confirm the equivalence of program variants [103, 104]. When a program variant successfully passes the test suite, it is deemed equivalent to the original. It is reasonable to assume that projects prioritizing quality and security are likely to have a robust test suite that facilitates this type of equivalence checking. However, this technique's effectiveness is limited by the necessity for a preexisting test suite. Yet, as an alternative, fuzzers can be used to automatically generate tests [105]. Fuzzers operate by randomly generating inputs that lead to different observable behaviors. If a variant produces a different output from two identical inputs, it is not equivalent to the original program. Fuzzers' primary drawback is their time-consuming nature and the requirement for manually introducing oracles. Recent advancements in the field of machine learning have led researchers to explore the application of neural networks in verifying program equivalence. Zhang and his team's work provides an example of this, where Large Language Models are used to generate reference oracles and test cases [106]. Despite its effectiveness, this method attains an accuracy rate of just 88%, which falls short of providing complete verification.

Formal verification: In the absence of a test suite or a technique that inherently implements the equivalence property, the works mentioned earlier use automated theorem provers. Theorem provers rely on SMT solvers [107] to prove the equivalence of program variants. The central idea for theorem provers is to convert the two code variants into mathematical formulas. The core component, the SMT solver, then checks for counter-examples that satisfy the negation of the mathematical formulas [108]. When it finds a counter-example, it uncovers an input for which the two mathematical formulas yield different outputs. The primary limitation of this technique resides in the conversion process. All algorithms can be translated into a mathematical formula. However, under certain theories such as loops for linear arithmetic, the satisfiability query may be undecidable. As a result, SMT solvers cannot make a decision. Nevertheless, this technique is frequently used for checking no-jump-programs like basic block and peephole replacements [109].

2.2.3 Variants deployment

Program variants, once generated and verified, may be used in two primary scenarios: Randomization or Multivariant Execution (MVE) [73].

Randomization: In the context of our work, the term *Randomization* denotes a program's ability to present different variants to different clients. In this setup, a program, randomly chosen from a collection of variants (referred to as the program's variant pool), is assigned to a the client during each deployment. Jackson et al. [73] define the variant pool in Randomization as herd immunity, as vulnerable binaries can only affect a segment of the client community. El-Khalil et al. [110] suggest employing a custom compiler to generate varying binaries from the compilation process. They adapt a version of GCC 4.1 to partition a conventional stack into several component parts, termed multistacks. Similarly, Singhal et al., propose Cornucopia [111]. Cornucopia generates multiple variants of a program by using different compiler flag combinations. Aga et al. propose the generation of program variants through the randomization of its data layout in memory[88]. This method allows each variant to operate on the same data in memory but at different memory offsets. Randomization can also be applied to virtual machines and operating systems. On this note, Kc et al. [92] establish a unique mapping between artificial CPU instructions and actual ones, enabling the assignment of various variants to specific target clients. In a similar vein, Xu et al. [91] recompile the Linux Kernel to minimize the exposure time of persistent memory objects, thereby increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed into a single binary, known as a multivariant binary [112]. Each multivariant binary is randomly deployed to a client. Then, the multivariant binary executes its embedded program variants at runtime. These embedded variants can either execute in parallel to check for inconsistencies, or as a single program to randomize

execution paths [83]. Bruschi et al. extend the concept of executing two variants in parallel, introducing non-overlapping and randomized memory layouts [113]. At the same time, Salamat et al. modify a standard library to generate 32-bit Intel variants. These variants have a stack that grows in the opposite direction, allowing for the detection of memory inconsistencies [114]. Davi et al. propose Isomeron, an approach for execution-path randomization [115]. Isomeron operates by simultaneously loading the original program and a variant. It then uses a coin flip to determine which copy of the program to execute next at the function call level. Previous works have highlighted the benefits of limiting execution to only two variants in a multivariant environment. Agosta et al., as well as Crane et al., used more than two generated programs in the multivariant composition, thereby randomizing software control flow at runtime [116, 86]. Both strategies have proven effective in enhancing security by addressing known vulnerabilities, such as Just-In-Time Return-Oriented Programming (JIT-ROP) attacks [117] and power side-channel attacks [118]. Lastly, only Voulimeneas et al. [119] have recently proposed a multivariant execution system that enhances security by parallelizing the execution of variants across different machines.

2.2.4 Measuring Software Diversification

Measuring Software Diversification presents a significant challenge. The size of the variant space does not necessarily correlate with a variant's capacity to fulfill an objective such as hardening attacks by making systems less predictable [33]. Ideally, real scenarios would provide the most accurate measurement of diversification, e.g., demonstrating a variant's effectiveness under specific attacks. However, such an approach is not always feasible, since Software Diversification is a preventive strategy. Hence, a combination of static and dynamic metrics is required for measuring Software Diversification.

Static comparison of variants: Static metrics are used to measure the diversity of programs without needing execution. The fundamental concept entails comparing variant source codes or binary codes to determine how diverse they are. Usually, comparing variants means defining a distance metric between programs [102] where the more different the programs are, the greater the distance. At the low-level of bytecode instructions, for example, these metrics include counting instructions [120], Levenshtein distance [121], and global alignments [36]. On the other hand, at the high-level of source code, these metrics often rely on Abstract Syntax Tree (AST) diffing, such as GUMtree-based distances [122] or machine learning inference [123]. As an example of measuring the diversification, Bostani et al. [124] illustrate the use of static distances in guiding the generation process of variants. They categorize the space of Android applications into malware and goodware. Then, they create malware variants by employing a static distance metric to approach the goodware group as closely as possible, thus successfully evading malware classifiers.

Dynamic comparison of variants: Static comparisons between variants inherently have limitations. For example, two variants may show differences at the source code level but exhibit identical behavior during execution. Take the addition of `nop` operations to a program as an instance. Despite source code level differences, the variant and the original program execute identical instructions, leading to similar behaviors modulo input. Measuring Software Diversification primarily aims to demonstrate variant-specific observabilities. While static differences are usually observable, runtime information holds complementary relevance [125]. Therefore, dynamic metrics are essential to assess the diversity of variants. For instance, Forrest et al. [126] were pioneers in classifying program behaviors by analyzing their system call traces using n-grams profiling. Cabrera et al. used a global alignments approach to gauge the diversity of JavaScript bytecode traces within the Chrome browser [14]. Fang et al. proposed a method to counteract JavaScript obfuscation techniques used in malicious code, by analyzing dynamic information captured from V8 bytecode traces [127]. Dynamic metrics are primarily employed to cluster similar behaviors. Following the same logic, the diversity is greater when the difference between behaviors is larger. Notice that, dynamic metrics can be difficult due to the expense of program execution or the complication of required user interaction. On the other hand, malware programs, which usually do not require user interaction, are simpler to evaluate in controlled environments before actual deployment.

In the context of WebAssembly, there exist no explicit works on Software Diversification. Consequently, previous metrics have not been directly applied to measure diversification in WebAssembly binaries. However, in other domains, such as the analysis of WebAssembly binaries, several studies have employed static metrics. For example, VeriWasm quantifies attack-based patterns, stating that a WebAssembly binary is more secure with a lower pattern count [128]. This metric might potentially serve as a guide during variant generation. In the field of malware detection, MINOS [28] proposes transforming WebAssembly binaries into grayscale images. They then employ convolutional neural networks to identify malware, where an increased similarity to a malware image increases the probability of the binary being malware. Regarding the dynamic comparisons, Wang et al.'s study [29] profiles WebAssembly instructions during runtime to identify malicious behavior.

2.2.5 Offensive or Defensive assessment of diversification

Lundquist et al. [75] distinguish Software Diversification into two categories: Defensive and Offensive Diversification. On the one hand, Defensive Software Diversification introduces unpredictability in system behavior. By making software less predictable, defensive Software Diversification aims to proactively deter attacks, acting as a complementary strategy to other, more reactive, security measures. The majority of previously discussed works in this section contribute to

defensive diversification. Yet, Software Diversification that aims to create diverse harmful programs is considered Offensive Diversification [129].

Offensive Diversification: Offensive Diversification is conceptually equal to Defensive Software Diversification. Yet, in an offensive context, one may apply diversification techniques to malware or other malicious codes to evade detection by security software [130]. One might equate Offensive Diversification with Code obfuscation, if its purpose shifts from preventing reverse engineering by malicious actors, to evading detection by malware analysis systems.

Malicious actors may employ previously discussed diversification strategies to evade detection [131]. For instance, in the Web context, Weihang et al. propose to randomly transform HTML elements of web pages to evade advertisement blockers [132]. Over time, evasion techniques have evolved in both complexity and sophistication [133]. Chua et al. [134], for instance, suggested a framework for automatically obfuscating the source code of Android applications using method overloading, opaque predicates, try-catch, and switch statement obfuscation, resulting in multiple versions of identical malware. Moreover, machine learning approaches have been used to develop evasive malware [135], drawing on a corpus of pre-existing malware [124]. These methods aim to thwart static malware detectors, yet, more advanced techniques focus on evading dynamic detection mostly by employing throttling [136, 137].

The term Offensive Software Diversification may seem counterintuitive. Yet, such approaches measure the resilience and accuracy of security systems. This is an almost unexplored area in WebAssembly, posing a threat to malware detection accuracy. Specifically, only Bhansali et al.'s seminal work[67] has demonstrated that a cryptomining algorithm's source code can evade pre-existing malware detection methods. More recently, Madvex [138] has sought to obfuscate WebAssembly binaries to achieve malware evasion, but this approach is limited to altering only the code section of WebAssembly binaries.

2.3 Open challenges for Software Diversification

As outlined in Section 2.1.6, our primary motivation for the contributions of this thesis are the open issues within the WebAssembly ecosystem. We see potential in employing Software Diversification to address them. Based on our previous discussion, we highlight several open challenges in the realm of Software Diversification for WebAssembly. First, WebAssembly, being an emerging technology, is in the process of implementing defensive measures. In addition, while measures for WebAssembly can be standardized, the implementation of these standards across the ecosystem is naturally slow. Therefore, applying Software Diversification directly to the generation of WebAssembly binaries, according to any given specification, could serve as a valuable strategy to lessen the impact of vulnerabilities. Second, despite the abundance of related work

on software diversity, its exploration in the context of WebAssembly remains limited. This thesis is the first to investigate Software Diversity in depth for the emerging WebAssembly ecosystem. Third, both randomization and multivariant execution remain largely unexplored within the WebAssembly context. The deployment of Software Diversification in WebAssembly poses unique challenges. WebAssembly ecosystems are remarkably dynamic. Web browsers and FaaS platforms serve as prime examples. In these environments, WebAssembly binaries are served millions of times simultaneously to the former, while new WebAssembly binaries are cold-spawned and executed upon each user request in the latter. Thus, designing practical Software Diversification for WebAssembly requires careful consideration of the deployment environment. Last but not least, research on malware detection, as discussed in Section 2.1.5, suggests that offensive diversification may assist in evaluating the resilience and accuracy of WebAssembly's security systems.

3
AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.

— David Wheeler

The process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components [139]: a frontend that converts the source code into an intermediate representation, a transformer that modifies this representation usually for performance, and a backend that compiles the final WebAssembly binary. This architecture is illustrated in the leftmost part of Figure 3.1.

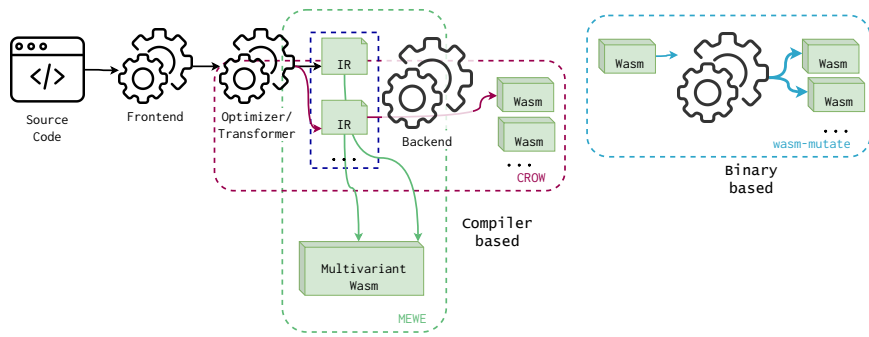


Figure 3.1: *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

Software Diversification can be integrated at various stages of this compilation process. However, applying diversification at the frontend has limitations, as it

would need a unique diversification mechanism for each language compatible with the frontend component. This makes the latter stages of the compilers an ideal point for introducing practical Wasm diversification techniques.

Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the transformer and backend stages. This transformer component generates variants in the intermediate representation of a compiler, thereby creating artificial software diversity for WebAssembly. The variants are then compiled into WebAssembly binaries by the backend component of the compiler. Concretely, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE, which packages these variants to enable multivariant execution [112]. Alternatively, diversification can be directly applied to the WebAssembly binary, offering a language and compiler-agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

Within this chapter, we introduce three technical contributions: CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility of our main takeaways.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [36], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combination*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combination* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

3.1.1 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [98]. The search space for generating variants is delineated through an enumerative synthesis process (see Enumerative synthesis

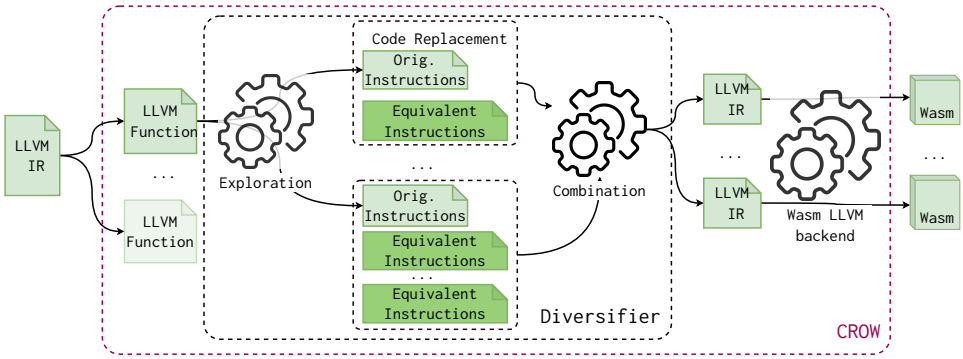


Figure 3.2: *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [40].*

in Section 2.2.1), which systematically produces all possible code replacements for each data flow graph that can be constructed from the original program. If a code replacement is identified to function identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

CROW is developed by extending the enumerative synthesis implementation found in Souper [140], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph(DFG) for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

CROW is carefully designed to boost the generation of variants as much as possible. First, we disable the majority of the pruning strategies. Instead of preventing the generation of commutative operations during the search, CROW still uses such transformation as a strategy to generate program variants. Second, CROW applies code transformations independently. For instance, if a suitable replacement is identified that can be applied at N different locations in the original program, CROW will generate 2^N distinct program variants, i.e., the power set of applying the transformation or not to each location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases. Third, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations

Notice that, the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm backend, effectively reducing the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter.

3.1.2 Constant inferring

CROW inherently introduces a transformation strategy called *constant inferring*, which significantly expands the variety of WebAssembly program variants. Specifically, CROW identifies segments of code that can be simplified into a single constant assignment, with a particular focus on variables that control branching logic. After applying this *constant inferring* technique, the resulting program diverges substantially from the original program structure. This is crucial for diversification efforts, as one of the primary objectives is to create variants that are as distinct as possible from the original code (see Section 2.2.4). In essence, the more divergent the variant, the more challenging it becomes to trace it back to its original form.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
            ↪ current);
8       return 0 ;
9   }
```

Listing 3.1: *Babbage problem. Taken from [40].*

```
int babbage() {
    int current = 25264;
    printf ("The number is %d\n", current)
        ↪ ;
    return 0 ;
}
```

Listing 3.2: *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [40].*

CROW deals with this case, generating the program in Listing 3.2. It infers

the value of `current` in Line 2 such that the Babbage condition is reached [1]. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[2].

3.1.3 Exemplifying CROW

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the leftmost part of Listing 3.4. CROW potentially finds two integers returning instructions to look for variants, as the rightmost part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

Listing 3.3: *C function that calculates the quantity* $2x + x$.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}

define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

[1] In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large.

[2] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the leftmost part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the rightmost parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

Listing 3.6: *Wasm program variants generated from program Listing 3.3.*

Contribution paper and artifact

CROW is a compiler-based approach. It leverages enumerative synthesis to generate functionally equivalent code replacements and assembles them into diverse Wasm program variants. CROW uses SMT solvers to guarantee functional equivalence.
CROW is fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of Measurements, Attacks, and Defenses for the Web (MADWeb), NDSS 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [37], our second technical contribution. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [83]. Execution path randomization is a technique that randomizes the execution path of a program at runtime, i.e. at each invocation of a function, a different variant is executed. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary. Creating a new entrypoint is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [41] for the random behavior of the dispatchers. However, its exact implementation is dependent on the host engine on which the binary is executed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

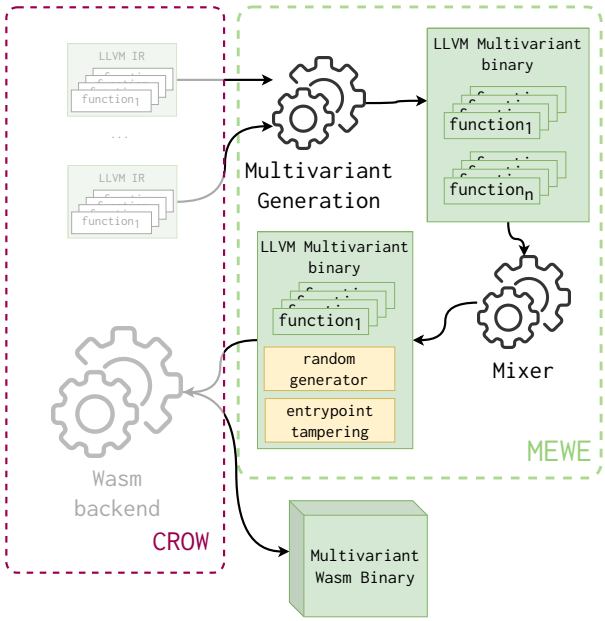


Figure 3.3: *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed.*

3.2.1 Multivariant call graph

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1 *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[141]. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

3.2.2 Exemplifying a Multivariant binary

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

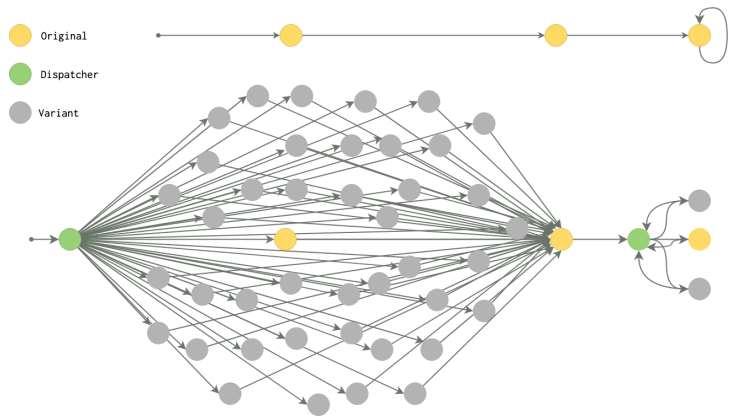


Figure 3.4: *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow).*

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [142] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [128]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect

calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

```
; Multivariant foo wrapping ;
define internal i32 @foo(i32 %0) {
    entry:
      ; It first calls the dispatcher to discriminate between the created
          variants ;
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    ;One case for each generated variant of foo ;
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      ; MEWE can inline the body of the a function variant ;
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      ; The original is also included ;
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented on for the sake of understanding.*

Contribution paper and artifact

MEWE provides dynamic execution path randomization by packaging variants generated out of CROW.
MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of Moving Target Defense, 2022, ACM* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [38], a tool that generates thousands of functionally equivalent variants out of a WebAssembly binary input. Leveraging rewriting rules and e-graphs [143]

for software diversification, WASM-MUTATE synthesizes program variants by transforming any section of the original WebAssembly binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

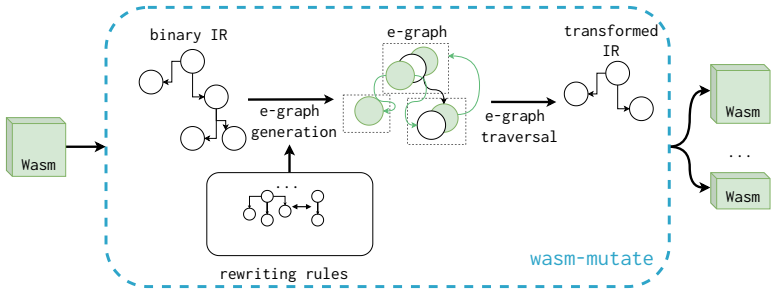


Figure 3.5: *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE uses predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. The assurance of functional equivalence is rooted in the inherent properties of the individual rewrite rules employed. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary.

WASM-MUTATE applies one transformation at a time. Notice that, the output of one applied transformation can be chained again as an input WebAssembly binary, enabling the generation of many variants, leading us to enunciate the notion of *Stacked transformation*

Definition 2 *Stacked transformation: Given an original input WebAssembly binary I and a diversifier D, stacked transformations are defined as the application of D over the binary I multiple times, i.e., $D(D(D(...(I))))$. Notice that, the number of stacked transformations is the number of times the diversifier D is applied.*

3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. In this context, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of binary targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the

replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [3].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program.

In the following example, we demonstrate a rewriting rule within the `Peephole` meta-rule that uses a custom `rand` operator to expand statically declared constants within any WebAssembly program function body. The unfolding rewriting rule, as the name suggests, transforms statically declared constants into the sum of two random numbers. During the generation of the WebAssembly variant, the custom `rand` operator is substituted with a randomly chosen static constant. Notice that the condition specified in the last part of the rewriting rule ensures that this predicate is satisfied.

```
LHS i32.const x

---

RHS (i32.add (i32.rand i32.const y))

Cond y = x - i32.rand
```

Although this rewriting approach may seem simplistic, especially because compilers often eliminate it through *Constant Folding* optimization [48], it stresses the spill/reload component of the compiler when the WebAssembly binary is JITed to machine code. Spill/reloads occur when the compiler runs out of physical registers to store intermediate calculations, resorting to specific memory locations for temporary storage. The unfolding rewriting rule indirectly stresses this segment of memory when applied many times to the input WebAssembly binary. Notably, with this specific rewriting rule, we have found a CVE in the wasmtime standalone engine [144].

[3] For an in-depth explanation of the remaining meta-rules, refer to [38].

3.3.2 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [145]. In an e-graph, there are two types of nodes: e-nodes and e-classes. An e-node represents either an operator or an operand involved in the rewriting rule, while an e-class denotes the equivalence classes among e-nodes by grouping them, i.e., an e-class is a virtual node compound of a collection of e-nodes. Thus, e-classes contain at least one e-node. Edges within the graph establish operator-operand equivalence relations between e-nodes and e-classes. In the context of WASM-MUTATE, e-graphs are constructed from the input WebAssembly program and the implemented rewriting rules (we detail the e-graph construction process in Section 3 of [38]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [143]. For instance, e-graphs address the challenge of generating the optimal code from multiple optimization rules, regardless of their application sequence [146]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to rapidly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is selected from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

3.3.3 Exemplifying WASM-MUTATE

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule grants the

Algorithm 1 e-graph traversal algorithm taken from [38].

```
1: procedure TRAVERSE(egraph, eclass, depth)
2:     if depth = 0 then
3:         return smallest_tree_from(egraph, eclass)
4:     else
5:         nodes ← egraph[eclass]
6:         node ← random_choice(nodes)
7:         expr ← (node, operands = [])
8:         for each child ∈ node.children do
9:             subexpr ← TRAVERSE(egraph, child, depth − 1)
10:            expr.operands ← expr.operands ∪ {subexpr}
11:        return expr
```

ability to append a `nop` instruction to any subexpression, a well-known low-level diversification strategy [82]. The latter rule adds zero to any numeric value.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: *Wasm function.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: *Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.*

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr = i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is selected, selecting the

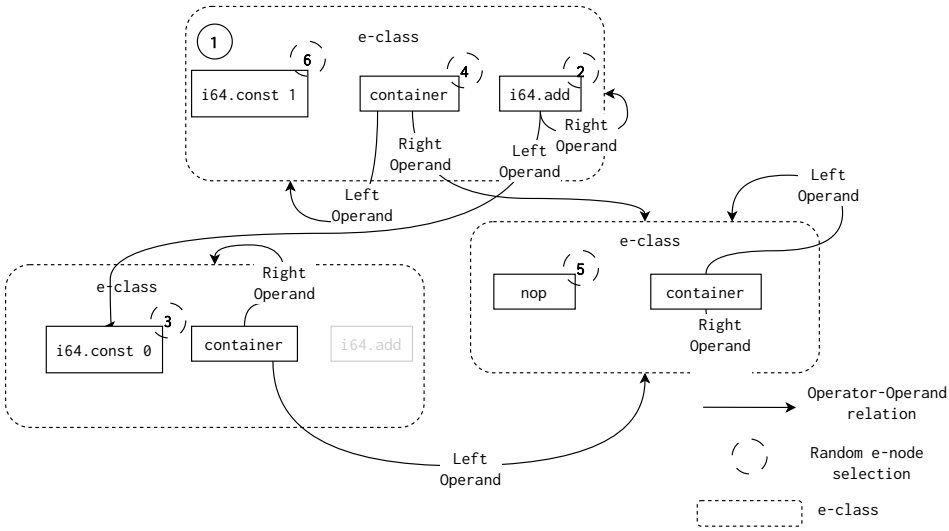


Figure 3.6: *e-graph built for rewriting the first instruction of Listing 3.8.*

`container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE uses hand-made rewriting rules and random traversals over e-graphs to provide a binary-based solution for WebAssembly diversification.
WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Submitted to Computers & Security, under revision.* `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the Bytecode Alliance organization [a]. The Bytecode Alliance is dedicated to creating secure new software foundations, building on standards such as WebAssembly and WASI.

[a] `https://bytecodealliance.org/`

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, the strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. The *Variant strength* accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime on average. For example, a higher value of the *Variant strength* indicates that the generated variants are not reversed by JIT compilers, ensuring that the diversification is preserved in an end-to-end scenario of a WebAssembly program, i.e. from the source code to its final execution. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

CROW is a compiler-based strategy that needs access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functional verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, using its generated variants. It goes a step further by packaging the LLVM IR variants into a WebAssembly multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code.

WASM-MUTATE, on the other hand, is a binary-based tool. It uses a set of rewriting rules and the input Wasm binary to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We have observed several interesting phenomena when aggregating the empirical data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [36, 37, 38]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because of three reasons. First, CROW and MEWE rely on SMT solvers to prove functional equivalence, placing a bottleneck when generating variants. Second, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Section 3.1.3. Third, WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections.

On the other hand, CROW and MEWE, by using enumerative synthesis, ensure that the generated variants are more preserved than the variants created by WASM-MUTATE. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE. High preservation is especially important where the preservation of the diversification is crucial, e.g. to hinder reverse engineering.

Tool	Input	Core	Variants in 1h	Target	Variants Strength	Security applications
CROW	Source code or LLVM Ir	Enumerative synthesis with functional equivalence proved through SMT solvers	> 1k	Code section	**96%**	Hinders static analysis and reverse engineering.
MEWE	Source code or LL VM Ir	CROW, Multivariant execution	> 1k	Code and Function sections	**96%**	Hinders static and dynamic analysis, reverse engineering and, web timing-based attacks.
WASM-MUTATE	**Wasm binary**	hand-made rewriting rules, e-graph random traversals	**> 10k**	**All WebAssembly sections**	76%	Hinders signature-based identification, and cache timing side-channel attacks.

Table 3.1: *Comparing CROW, MEWE and WASM-MUTATE. The table columns are the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, the strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

Takeaway

Our three technical contributions serve as complementary tools that can be combined. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial.

3.4.1 Security applications

The final column of Table 3.1 emphasizes the security benefits derived from the variants produced by our three key technical contributions. One immediate advantage of altering the structure of WebAssembly binaries across different variants is the mitigation of signature-based identification, thereby enhancing resistance to static reverse engineering. Additionally, our tools generate a diverse array of code variants that are highly preserved. This implies that these variants, each with their unique WebAssembly code, retain their distinct characteristics even after being translated into machine code by JIT compilers. This high level of preservation significantly mitigates the risks associated with side-channel attacks that target specific machine code instructions, such as port contention attacks [18]. For instance, if a WebAssembly binary is transformed in such a manner that its resulting machine code instructions differ from the original, it becomes more challenging for a side-channel attack. On the other hand, if the compiler translates the variant into machine code that closely resembles the original, the side-channel attack could still exploit those instructions to extract information about the original WebAssembly binary.

Any structural alteration of a WebAssembly program intrinsically impacts its managed memory during runtime. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential cache timing side-channels [63]. This impact bears significant relevance for CROW and MEWE as they do not directly address the WebAssembly memory model. Nevertheless, the WebAssembly code section undergoes significant modifications by CROW and MEWE. These changes substantially alter the managed memory by transforming the layout of the WebAssembly binary in memory once JITed. For example, the *constant inferring* transformations are "aggresive" since they considerably change the structure of a WebAssembly variant. Thus, they considerably affect unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout as CROW and MEWE do. It also adds explicit rewriting rules to transform unmanaged memory instructions.

Last but not least, our technical contributions enhance security against web timing-based attacks [147, 148] by creating variants that exhibit a wide range

of execution times, including faster variants compared to the original program. This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [148]. Adding another layer benefit from MEWE, the integration of diverse variants into multivariants can potentially disrupt dynamic reverse engineering tools such as symbolic executors [96]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

Takeaway

CROW, MEWE and WASM-MUTATE generate WebAssembly variants that can be used to enhance security. Overall, they generate variants that are suitable for hardening static and dynamic analysis, side-channel attacks, and, thwarting signature-based identification.

3.5 Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, proposing a novel e-graph traversal algorithm to fast spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In Chapter 4, we present two use cases that support the exploitation of these tools. Chapter 4 serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

4 ASSESSING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

If you find that you're spending all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.

— Donald Knuth

In this chapter, we illustrate the application of Software Diversification for both offensive and defensive purposes. We discuss two selected use cases that demonstrate the practical applications of our contributions. Additionally, we discuss the challenges and benefits arising from the application of Software Diversification to WebAssembly.

4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [66]. This is due to the essence of cryptojacking, the faster the mining, the better. Let us illustrate how a malicious WebAssembly binary is involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

The aforementioned components require several steps to succeed in cryptomining. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then

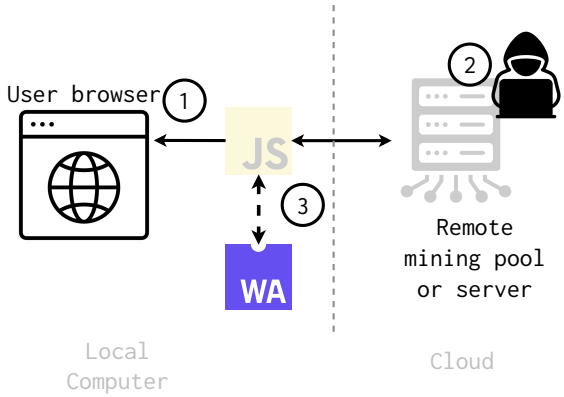


Figure 4.1: *A remote mining pool server, a JavaScript wrapper and the WebAssembly binary form the triad of a cryptojacking attack in browser clients.*

assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [149]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Section 2.1.5. However, malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [67], highlighting the critical importance of this use case. The case illustrated in the subsequent sections uses Offensive Software Diversification for evading malware detection in WebAssembly.

4.1.1 Cryptojacking defense evasion

Considering the previous scenario, several techniques can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such a defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user loads a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ②

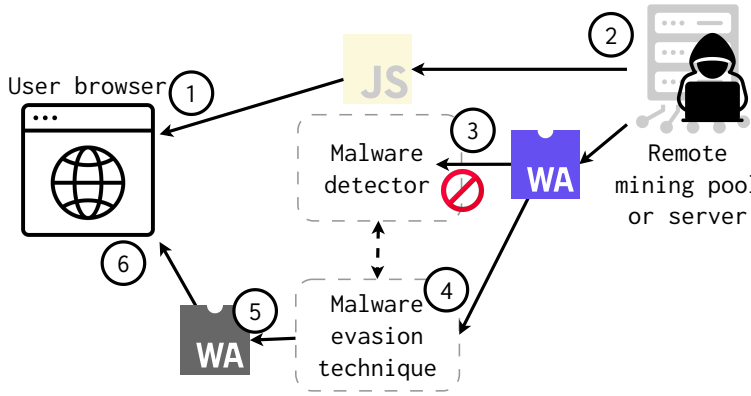


Figure 4.2: *Cryptojacking scenario in which the malware detection mechanism is bypassed by using an evasion technique.*

in Figure 4.2.

2. A malware detection mechanism (malware oracle) identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.

3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.

4. The attacker delivers the modified binary instead of the original one ⑤, which initiates the cryptojacking process and compromises the browser ⑥. The detection method is not capable of detecting the malicious nature of the binary, and the attack is successful.

4.1.2 Methodology

Our aim is to empirically validate the workflow in Figure 4.2, i.e., using Offensive Software Diversification in evading malware detection systems. To achieve this, we employ WASM-MUTATE for generating WebAssembly malware variants. In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

Definition 3 *Malware oracle: A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.*

We employ VirusTotal as a numeric oracle and MINOS [28] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score in the form of the number of antivirus that flag the input file as malware, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [45] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries that are: 1) flagged as malicious by at least one VirusTotal vendor and, 2) are also detected by MINOS. Then, to simulate the evasion scenario in Figure 4.2, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection (④ in Figure 4.2). We use WASM-MUTATE in two configurations: feedback-guided and stochastic diversification.

Definition 4 *Feedback-guided Diversification: In feedback-guided diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle's confidence score. Note that feedback-guided diversification needs a numeric oracle.*

Definition 5 *Stochastic Diversification: Unlike feedback-guided diversification, in stochastic diversification, each transformation has an equal likelihood of being applied to the input WebAssembly binary.*

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a feedback-guided diversification, 2) VirusTotal with a stochastic diversification, and 3) MINOS with a stochastic diversification. Notice that, the fourth scenario with MINOS and feedback-guided diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for feedback-guided diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the correctness of the generated variants.

Definition 6 *Evasion rate: This measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the oracle or when a maximum of 1000 stacked transformations have been applied (see Definition 2). This process is repeated with 10 random seeds per binary to*

simulate 10 different evasion experiments per binary.

Definition 7 *Correctness: This validates the functional equivalence of the variants generated by WASM-MUTATE compared to the original binary. We execute the variants that entirely evade VirusTotal, using controlled and stochastic diversification configurations with WASM-MUTATE for both metrics. Our selection is limited to variants that allow us to fully reproduce the three components displayed in Figure 4.1. We then gather the hashes generated by the cryptojacking binaries and their generation speed, comparing these hashes with those from the original binary. If the hashes match, and the variant executes without error, with the minerpool component validating the hash, we can consider the variant as functionally equivalent.*

4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [39], focusing on two oracles: VirusTotal and MINOS[28]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: stochastic and feedback-guided diversification. In these subsections, the columns indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the stochastic diversification or feedback-guided diversification setups best, the lower, the better. The MINOS section solely includes a column that specifies the number of transformations needed for complete evasion. The table has 33 + 1 rows, each representing a unique WebAssembly malware study subject. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

Stochastic diversification to evade VirusTotal: We execute a stochastic diversification with WASM-MUTATE, setting a limit of 1000 iterations for each binary. In every iteration, we query VirusTotal to determine if the newly generated binary can elude detection. We repeat this procedure with ten distinct seeds for each binary, replicating ten different evasion experiments. As the stochastic diversification section of Table 4.1 illustrates, we successfully produce variants that fully evade detection for 30 out of 33 binaries. The average amount of iterations required to produce a variant that evades all detectors oscillates between 120 and 635 stacked transformations. The mean number of iterations needed never exceeds 1000 stacked transformations. However, three binaries remain detectable under the stochastic diversification setup. In these instances, the algorithm fails to evade 5 out of 31, 6 out of 30, and 5 out of 26 detectors. This shortfall can be attributed to the maximum number of iterations, 1000, that we employ in our experiments. Increasing iterations further, however, seems

	VirusTotal					MINOS[28]
Hash	#D	Stochastic diversification		Feedback-guided diversification		
		Max. evaded	Mean trans.	Max. evaded	Mean trans.	Mean trans.
47d29959	31	**26**	N/A	19	N/A	100
9d30e7f0	30	**24**	N/A	17	N/A	419
8ebf4e44	26	**21**	N/A	13	N/A	92
c11d82d	20	20	**355**	20	446	115
0d996462	19	19	**401**	19	697	24
a32a6f4b	18	18	635	18	**625**	1
fbdd1efa	18	18	**310**	18	726	1
d2141ff2	9	9	**461**	9	781	81
aafff587	6	6	484	6	**331**	1
046dc081	6	6	404	6	**159**	33
643116ff	6	6	**144**	6	436	47
15b86a25	4	4	253	4	**131**	1
006b2fb6	4	4	**282**	4	380	1
942be4f7	4	4	200	4	200	29
7c36f462	4	4	236	4	**221**	85
fb15929f	4	4	**297**	4	475	1
24aae13a	4	4	**252**	4	401	980
000415b2	3	3	302	3	**34**	960
4cbdbbb1	3	3	295	3	**72**	1
65debcbe	2	2	131	2	**33**	38
59955b4c	2	2	130	2	**33**	38
89a3645c	2	2	431	2	**107**	108
a74a7cb8	2	2	124	2	**33**	38
119c53eb	2	2	104	2	**18**	1
089dd312	2	2	153	2	**123**	68
c1be4071	2	2	130	2	**33**	38
dceaf65b	2	2	140	2	132	66
6b8c7899	2	2	143	2	**33**	38
a27b45ef	2	2	145	2	**33**	33
68ca7c0e	2	2	137	2	**33**	38
f0b24409	2	2	127	2	**11**	33
5bc53343	2	2	118	2	**33**	33
e09c32c5	1	1	**120**	1	488	15
Median			218		131	38

Table 4.1: *The table has two main categories for each malware oracle, corresponding to the two oracles we use: VirusTotal and MINOS. For VirusTotal, divide the results based on the two diversification configurations: stochastic and feedback-guided diversification. We provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which diversification setups are best, the lower, the better. The MINOS section includes a column that specifies the number of transformations needed for complete evasion. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.*

unrealistic. If certain transformations enlarge the binary size, a significantly large binary could become impractical due to bandwidth limitations. In summary, stochastic diversification with WASM-MUTATE markedly reduces the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion in 30 out of 33 (90%) cases within the malware dataset.

Feedback-guided diversification to evade VirusTotal: stochastic diversification does not guide the diversification based on the number of evaded detectors, it is purely random and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the feedback-guided diversification part of Table 4.1. The feedback-guided diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it is thus as good as the stochastic setup. Remarkably, for 21 binaries out of 30, feedback-guided needs only 40% of the calls the stochastic diversification setup needs, demonstrating larger efficiency. Moreover, the lower number of transformations needed to evade detection, compared to the stochastic diversification setup, highlights the efficacy of the feedback-guided diversification setup in studying effective transformations. Consequently, malware detection system developers can leverage feedback-guided diversification to enhance their systems, focusing on identifying specific transformations.

Stochastic diversification to evade MINOS: Relying exclusively on VirusTotal for detection could pose issues, particularly given the existence of specialized solutions for WebAssembly, which differ from the general-purpose vendors within VirusTotal. In Section 2.1.5 we highlight several examples of such solutions. Yet, for its simplicity, we extend this experiment by using MINOS[28], an antivirus specifically designed for WebAssembly. The results of evading MINOS can be seen in the final column of Table 4.1. The bottom row of Table 4.1 highlights that fewer iterations are required to evade MINOS than VirusTotal through WebAssembly diversification, indicating a greater ease in eluding MINOS. The stochastic diversification setup requires a median iteration count of 218 to evade VirusTotal. In contrast, the feedback-guided diversification setup necessitates only 131 iterations. Remarkably, a mere 38 iterations are needed for MINOS. WASM-MUTATE evaded detection for 8 out of 33 binaries in a single iteration. This result implies the susceptibility of the MINOS model to binary diversification.

WebAssembly variants correctness: To evaluate the correctness of the malware variants created with WASM-MUTATE, we focused on six binaries that we could build and execute end-to-end, as these had all three components outlined

in Figure 4.1. We select only six binaries because the process of building and executing the binaries involves three components: the WebAssembly binary, its JavaScript complement, and the miner pool. These components were not found for the remaining 24 evaded binaries in the study subjects. For the six binaries, we then replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and stochastic diversification configurations. We then execute both the original and the generated variants. We assess the correctness of the variants by examining the hashes they generate. Our findings show that all variants generated with WASM-MUTATE are correct, i.e., they generate the correct hashes and execute without error. Additionally, we found that 19% of the generated variants surpassed the original cryptojacking binaries in performance.

Reflection

Malware detection presents a challenging and well-known issue [150]. While there are considerable efforts on preventing malware in WebAssembly, the current literature acknowledges only metadata (WebAssembly custom sections) obfuscation or total absence of obfuscation techniques for WebAssembly [26, 27, 29, 30, 28]. As explored in Section 2.2, a software diversification engine could potentially serve as an obfuscator. We exhibit this potential with WASM-MUTATE. Moreover, our software diversification tools offer a feasible method to improve the precision of WebAssembly malware detection systems. Existing tools could enhance their evaluation dataset of WebAssembly malware by incorporating the variants generated by WASM-MUTATE.

Contribution paper

WASM-MUTATE generates correct and performant variants of WebAssembly cryptojacking that successfully evade malware detection. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

4.2 Defensive Diversification: Speculative Side-channel protection

As discussed in Section 2.1, WebAssembly is quickly becoming a cornerstone technology in backend systems. Leading companies like Cloudflare and Fastly are championing the integration of WebAssembly into their edge computing platforms, thereby enabling developers to deploy applications that are both modular and securely sandboxed. These server-side WebAssembly applications

are generally architected as isolated, single-responsibility services, a model referred to as Function-as-a-Service (FaaS) [10, 11]. The operational flow of WebAssembly binaries in FaaS platforms is illustrated in Figure 4.3.

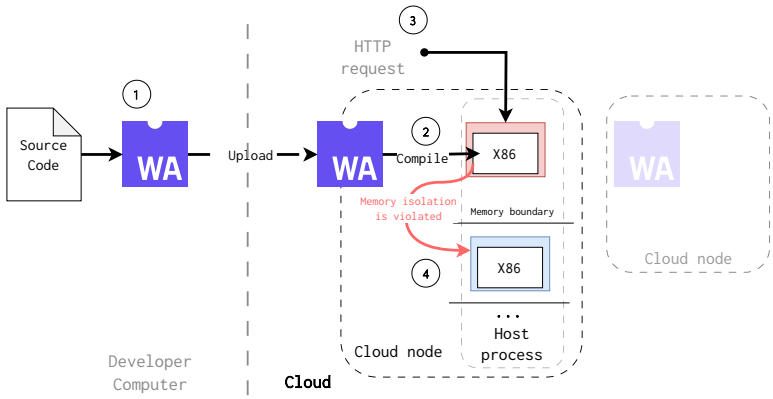


Figure 4.3: *WebAssembly binaries on FaaS platforms. Developers can submit any WebAssembly binary to the platform to be executed as a service in a sandboxed and isolated manner. Yet, WebAssembly binaries are not immune to Spectre attacks.*

The fundamental advantage of using WebAssembly in FaaS platforms lies in its ability to encapsulate thousands of WebAssembly binaries within a singular host process. A developer could compile its source code into a WebAssembly program suitable for the cloud platform and then submit it (① in Figure 4.3). This host process is then disseminated across a network of servers and data centers (② in Figure 4.3). These platforms convert WebAssembly programs into native code, which is subsequently executed in a sandboxed environment. Host processes can then instantiate new WebAssembly sandboxes for each client function, executing them in response to specific user requests with nanosecond-level latency (③ in Figure 4.3). This architecture inherently isolates WebAssembly binary executions from each other as well as from the host process, enhancing security.

However, while WebAssembly is engineered with a strong focus on security and isolation, it is not entirely immune to vulnerabilities such as Spectre attacks [151, 142] (④ in Figure 4.3). In the sections that follow, we explore how software diversification techniques can be employed to harden WebAssembly binaries against such attacks.

4.2.1 Threat model: speculative side-channel attacks

To illustrate the threat model concerning WebAssembly programs in FaaS platforms, consider the following scenario. Developers, including potentially malicious actors, have the ability to submit any WebAssembly binary to FaaS platforms. A malicious actor could then upload a WebAssembly binary that,

once compiled to native code, employs Spectre attacks. Spectre attacks exploit hardware-based prediction mechanisms to trigger mispredictions, leading to the speculative execution of specific instruction sequences that are not part of the original, sequential execution flow. By taking advantage of this speculative execution, an attacker can potentially access sensitive information stored in the memory allocated to another WebAssembly instance (including itself by violating Control Flow Integrity) or even the host process. Therefore, this poses a significant risk for the overall execution system.

Narayan et al. [142] have categorized potential Spectre attacks on WebAssembly binaries into three distinct types, each corresponding to a specific hardware predictor being exploited and a particular FaaS scenario: Branch Target Buffer Attacks, Return Stack Buffer Attacks, and Pattern History Table Attacks defined as follows:

1. The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target.

2. The Spectre Return Stack Buffer (rsb) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions.

3. The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition.

4.2.2 Methodology

Our goal is to empirically validate that Software Diversification can effectively mitigate the risks associated with Spectre attacks in WebAssembly binaries. The green-highlighted section in Figure 4.4 illustrates how Software Diversification can be integrated into the FaaS platform workflow. The core idea is to generate unique and diverse WebAssembly variants that can be randomized at the time of deployment. For this use case, we employ WASM-MUTATE as our tool for Software Diversification.

To empirically demonstrate that Software Diversification can indeed mitigate Spectre vulnerabilities, we reuse the WebAssembly attack scenarios proposed by Narayan et al. in their work on Swivel [63]. Swivel is a compiler-based strategy designed to counteract Spectre attacks on WebAssembly binaries by linearizing their control flow during machine code compilation. Our approach differs from theirs in that it is binary-based, compiler-agnostic, and platform-agnostic; we do not propose altering the deployment or toolchain of FaaS platforms.

To measure the efficacy of WASM-MUTATE in mitigating Spectre, we diversify four WebAssembly binaries proposed in the Swivel study. The names of these programs and the specific attacks we examine are available in Table 4.2.

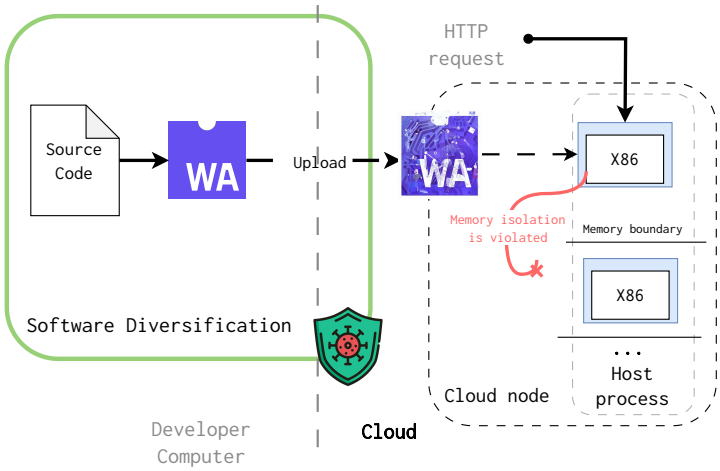


Figure 4.4: *Diversifying WebAssembly binaries to mitigate Spectre attacks in FaaS platforms.*

Program	Attack
btb_breakout	Spectre branch target buffer (btb)
btb_leakage	Spectre branch target buffer (btb)
ret2spec	Spectre Return Stack Buffer (rsb)
pht	Spectre Pattern History Table (pht)

Table 4.2: *WebAssembly program name and its respective attack.*

For each of these four binaries, we generate up to 1000 random stacked transformations (see Definition 2) using 100 distinct seeds, resulting in a total of 100,000 variants for each original binary. At every 100th stacked transformation for each binary and seed, we assess the impact of diversification on the Spectre attacks by measuring the attack bandwidth for data exfiltration.

Definition 8 *Attack bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* T *and* $K = k_0, k_1, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

The previous metric not only captures the success or failure of the attacks but also quantifies the extent to which data exfiltration is hindered. For example, a variant that still leaks data but does so at an impractically slow rate would be considered hardened against the attack.

4.2.3 Results

Figure 4.5 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs: btb_breakout and btb_leakage with the btb attack. The Y-axis represents the exfiltration bandwidth (see Definition 8). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by the green violinplots.

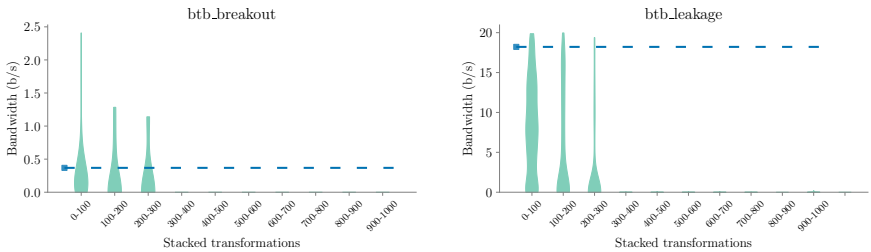


Figure 4.5: *Impact of WASM-MUTATE over btb_breakout and btb_leakage binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth. After 200 stacked transformations, the exfiltration bandwidth drops to zero.*

Population Strength: For the binaries btb_breakout and btb_leakage, WASM-MUTATE exhibits a high level of effectiveness, generating variants that leak less information than the original in 78% and 70% of instances, respectively. For both programs, after applying 200 stacked transformations, the exfiltration bandwidth drops to zero. This implies that WASM-MUTATE is capable of synthesizing variants that are entirely protected from the original attack. If we consider the results in Table 3.1, generating a variant with 200 stacked transformations can be accomplished in just a matter of seconds for a single WebAssembly binary.

Effectiveness of WASM-MUTATE: As illustrated in Figure 4.6, similarly to Figure 4.5, WASM-MUTATE significantly impacts the programs ret2spec and pht when subjected to their respective attacks. In 76% of instances for ret2spec and 71% for pht, the generated variants demonstrated reduced attack bandwidth compared to the original binaries. The plots reveal that a notable decrease in exfiltration bandwidth occurs after applying at least 100 stacked transformations. While both programs show signs of hardening through reduced attack bandwidth, this effect is not immediate and requires a substantial number of transformations to become effective. Additionally, the bandwidth distribution is more varied for

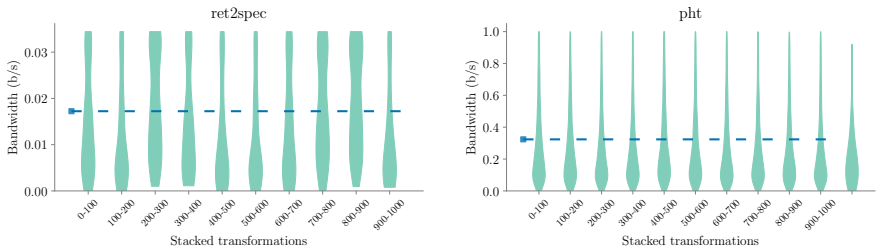


Figure 4.6: *Impact of WASM-MUTATE over ret2spec and pht binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for both programs approximately 70% of the variants have less data leakage bandwidth.*

these two programs compared to the two previous ones. Our analysis suggests a correlation between the reduction in attack bandwidth and the complexity of the binary being diversified. Specifically, ret2spec and pht are substantially larger programs, containing over 300,000 instructions, compared to btb_breakout and btb_leakage, which have fewer than 800 instructions. Therefore, given that WASM-MUTATE performs one transformation per invocation, the probability of affecting critical components to hinder attacks decreases in larger binaries.

Disrupting timers: Cache timing side-channel attacks, including for the four binaries analyzed in this use case, depend on precise timers to measure cache access times. Disrupting these timers can effectively neutralize the attack [152]. One key reason our results show variants resilient to Spectre attacks is the approach of WASM-MUTATE. It creates variants that offer a similar approach. Our WebAssembly variants introduce perturbations in the timing steps of WebAssembly variants. This is illustrated in Listing 4.1 and Listing 4.2, where the former shows the original time measurement and the latter presents a variant with introduced operations. By introducing additional instructions, the inherent randomness in the time measurement of a single or a few instructions is amplified, thereby reducing the timer's accuracy.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.1: *Wasm timer code.*

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.2: *WebAssembly variant with more instructions added in between time measurement.*

Padding speculated instructions: CPUs have a limit on the number of instructions they can cache. WASM-MUTATE injects instructions to exceed this limit, effectively disabling the speculative execution of memory accesses. This approach is akin to padding [153], as demonstrated in Listing 4.3 and Listing 4.4. Padding disrupts the binary code's layout in memory, hindering the attacker's ability to initiate speculative execution. Even if speculative execution occurs, the memory access does not proceed as the attacker intended.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

Listing 4.3: *Two jump locations. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.*

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

Listing 4.4: *WebAssembly variant with more instructions added indindinctly between jump places.*

Managed memory impact: The success in diminishing Spectre attacks is mainly explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. We have identified four primary factors responsible for the divergence in memory accesses among WASM-MUTATE generated variants. First, modifications to the binary layout—even those that do not affect executed code—inevitably alter memory accesses within the program's stack. Specifically, WASM-MUTATE generates variants that modify the return addresses of functions, which consequently leads to differences in execution flow and memory accesses. Second, one of our rewriting rules incorporates artificial global values into WebAssembly binaries. The access to these global variables inevitably affects the managed memory (see Section 2.1.3). Third, WASM-MUTATE injects 'phantom' instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the wasmtime compiler, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating diverse managed memory accesses (see the example at Section 3.3.1). Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does.

Reflection

Beyond Spectre, one can use WASM-MUTATE to mitigate other side-channel attacks. For instance, port contention attacks [18] rely on the execution of specific instructions for a successful attack. Not only WASM-MUTATE but also our other tools, can alter those instructions, thereby mitigating the attack [120]. The effectiveness of WASM-MUTATE, coupled with its ability to generate numerous variants, establishes it as an apt tool for mitigating side-channel attacks. Consider, for example, applying this on a global FaaS platform scale. In this scenario, one could deploy a unique, hardened variant for each machine and even for every fresh WebAssembly spawned per user request.

Contribution paper

WASM-MUTATE crafts WebAssembly binaries that are resilient to Spectre-like attacks. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *submitted to Computers & Security, under revision, 2023* `https://arxiv.org/pdf/2309.07638.pdf`.

4.3 Conclusions

In this chapter, we explore Offensive and Defensive Software Diversification applied to WebAssembly. Offensive Software Diversification highlights both the potential and the latent security risks in applying Software Diversification to WebAssembly malware. Our findings suggest potential enhancements to the automatic detection of cryptojacking malware in WebAssembly, e.g., by stressing their resilience with WebAssembly malware variants. Conversely, Defensive Software Diversification serves as a proactive guard, specifically designed to mitigate the risks associated with Spectre attacks.

Moreover, we have conducted experiments with various use cases that are not shown in this chapter. For instance, CROW [36] excels in generating WebAssembly variants that minimize side-channel noise, thereby bolstering defenses against potential side-channel attacks. Alternatively, deploying multivariants from MEWE [37] can thwart high-level timing-based side-channels [147]. Specifically, we conducted experiments on the round-trip times of the generated multivariants and concluded that, at a high level, the timing side-channel information cannot discriminate between variants.

5 CONCLUSIONS AND FUTURE WORK

You're bound to be unhappy if you optimize everything.

— Donald Knuth

THE growing adoption of WebAssembly requires hardening techniques. This thesis contributes to this effort with a comprehensive set of methods and tools for Software Diversification in WebAssembly. We introduce three technical contributions in this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

5.1 Summary of technical contributions

This thesis expands the field of Software Diversification within WebAssembly by implementing two distinct methods: compiler-based and binary-based approaches. Taking source code and LLVM bitcode as input, the compiler-based method generates WebAssembly variants. It uses enumerative synthesis and SMT solvers to produce numerous functionally equivalent variants. Importantly, these generated variants can be converted into multivariant binaries, thus enabling execution path randomization. Our compiler-based approach specializes in producing high-preservation variants. However, the use of SMT solvers for functional verification lowers the diversification speed when compared with the binary-based method. Furthermore, this method can only modify the code and function sections of WebAssembly binaries.

On the other hand, the method based on binary uses random e-graph traversals to create variants. This approach eliminates the need for modifications to existing compilers, ensuring compatibility with all existing WebAssembly binaries. Additionally, it offers a swift, efficient and novel method for generating variants through inexpensive random e-graph traversals. Consequently, our binary-based approach can produce variants at a scale of at least one order of

magnitude larger than our compiler-based approach. The binary-based method can generate variants by transforming any segment of the Wasm binary. However, the preservation of the generated variants is lower than the compiler-based approach.

We have developed three open-source tools that are publicly accessible: CROW, MEWE, and WASM-MUTATE. CROW and MEWE explore a compiler-based approach, whereas WASM-MUTATE employs a method based on binary. These tools automate the process of diversification, thereby increasing their practicality for deployment. At present, WASM-MUTATE is integrated into the wasmtime project[1] to improve testing. Our tools are complementary, providing combined utility. For instance, when the source code for a WebAssembly binary is unavailable, WASM-MUTATE offers an efficient solution for the generation of code variants. On the other hand, CROW and MEWE are particularly suited for scenarios that require a high level of variant preservation. Finally, one can use CROW and MEWE to generate a set of variants, which can then serve as rewriting rules for WASM-MUTATE. Moreover, when practitioners need to quickly generate variants, they could employ WASM-MUTATE, despite a potential decrease in the preservation of variants.

5.2 Key results of the thesis

We demonstrate the practical application of Offensive Software Diversification in WebAssembly. In particular, we diversify 33 WebAssembly cryptomalware automatically, generating numerous variants. These variants successfully evade detection by state-of-the-art malware detection systems. Our research confirms the existence of opportunities for the malware detection community to strengthen the automatic detection of cryptojacking WebAssembly malware. Specifically, developers can improve the detection of WebAssembly malware by using multiple malware oracles. Additionally, these practitioners could employ feedback-guided diversification to identify specific transformations their implementation is susceptible to. For instance, our study found that the addition of arbitrary custom sections to WebAssembly binaries is a highly effective transformation for evading detection. This logic also applies to other transformations, such as adding unreachable code, another effective method for evading detection.

Moreover, our techniques enhance overall security from a Defensive Software Diversification perspective. We facilitate the deployment of unique, diversified and hardened WebAssembly binaries. As previously demonstrated, WebAssembly variants produced by our tools exhibit improved resistance to side-channel attacks. Our tools generate variants by modifying malicious code patterns such as embedded timers used to conduct timing side-channel attacks. Simultaneously, they can produce variants that introduce noise into the execution side-channels

[1] `https://github.com/bytecodealliance/wasm-tools`

of the original program, while altering the memory layout of the JITed code generated by the host engine.

Remarkably, we ensure the rapid transformation of WebAssembly binaries, creating thousands of variants in a matter of minutes. This swift generation of variants is particularly advantageous in highly dynamic scenarios such as FaaS and CDN platforms. In this work, we do not discuss this case in depth. Yet, we have empirically tested the effectiveness of moving target defense techniques [117]. These tests were conducted on the Fastly edge computing platform. In this scenario, we incorporate multivariant executions[37]. Fastly can redeploy a WebAssembly binary across its 73 data centers worldwide in 13 seconds on average. This enables the practical deployment of a unique variant per node using our tools. However, a 13-second window may still pose a risk despite each node potentially hosting a distinct WebAssembly variant. To mitigate this, one could use multivariant binaries, invoking a unique variant every time the node is invoked. Our tools can generate dozens of unique variants every few seconds, each serving as a multivariant binary packaging thousands of other variants. This illustrates the real-world application of Defensive Software Diversification to a WebAssembly standalone scenario.

5.3 Future Work

Along with this dissertation, we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline three concrete directions.

5.3.1 Data augmentation for Machine Learning on WebAssembly programs

Compared to established environments, the WebAssembly ecosystem is relatively new. This makes the collection of WebAssembly program samples notably expensive [45, 154]. According to a recent study by Hilbig et al., the global count of WebAssembly binaries is approximately 20,000 [45]. This number is small when contrasted with the massive repositories of 1.5 million and 1.7 million packages in npm and PyPI, respectively. Intriguingly, this study also discloses that half of these WebAssembly binaries originated from our tools and public repositories, suggesting that the actual count of unique, real-world WebAssembly programs is just over 10000. This scarcity of WebAssembly datasets presents considerable obstacles for machine learning analysis tools, which typically need substantial data volumes for effective training and calibration [155]. Software diversification serves as a pivotal strategy to address this limitation by simulating a broad range of potential software behaviors and scenarios. Specifically, the augmentation of the WebAssembly dataset can be achieved through it.

Augmentation of program datasets has proven effective in enhancing the accuracy of classification models [156, 157, 158]. In general, data augmentation can improve model performance by increasing the volume of training examples via data synthesis. Moreover, it might expose edge cases and rare conditions, thus enabling the development of better defenses against unforeseen cases. In the case of malware evasion, this approach might harden the robustness of detection systems. Furthermore, this strategy might facilitate the identification and reduction of biases within WebAssembly program datasets. Finally, our tools provide comprehensive details on the variant generation process. This allows us to define and use more precise metrics between programs and variants to train a model for variant detection [159]. For instance, with the e-graphs in WASM-MUTATE, we can easily establish a metric to measure the distance between two programs. To sum up, we plan to harness our tools to enhance the training of models within the WebAssembly ecosystem.

5.3.2 Improving WebAssembly malware detection via canonicalization

Malware detection is a well-known problem in the field of computer security, as outlined in works like Cohen's 1987 study on computer viruses [150]. This issue is exacerbated in environments where predictability is high and malware is expected to be replicated identically across multiple victims. In such scenarios, attackers can exploit this predictability to their advantage. For example, malicious actors could craft functionally equivalent malware to evade detection by malware detection systems. Indeed, our research has shown that employing Software Diversification can be an effective method for evading malware detection systems. This technique involves creating varied versions of a program, thereby reducing its predictability and making detection more challenging.

In our future research, we plan to tackle the challenge of refining the precision of malware detection systems. We aim to achieve this by evaluating the effectiveness of program normalization [160]. This strategy involves transforming a program into a standardized or "canonical" form [161]. A malware detection system in a pre-existing database is more likely to detect the canonical form. Just as a program can be transformed into multiple functionally equivalent variants, the inverse process is also possible. In simple terms, two functionally equivalent programs can be transformed into the same original program.

We plan to examine two strategies. First, we aim to employ WASM-MUTATE for program normalization. By reusing the e-graph data structures to use the shortest possible replacements, we can secure the canonical representation of the input program. Although normalization methods are not new, previous studies have grounded malware detection on the normalization of lifted code [162, 163]. WebAssembly does not need to be lifted, given that its binary format is innately platform-agnostic. Thus, we can directly normalize the WebAssembly binary. Secondly, we can pre-compile WebAssembly binaries at a minimal cost. Specifically, a WebAssembly binary could initially undergo JIT compilation into

machine code. Overall, either of these two strategies aims to substantially reduce the number of malware variants that need consideration, thereby easing the task of classifiers in detecting harmful software.

5.3.3 Oneshot Diversification

Oneshot diversification denotes the automatic creation of a unique software program variant with each installation or distribution. This procedure entails systematic alterations to the original program. The objective is to ensure that each software copy is distinctive from all others. Contrary to randomization, oneshot diversification usually happens during the software's distribution or installation phase. The term "oneshot" describes the diversification's static nature following its one-time implementation per installation. Once used, the diversified program is discarded. In summary, oneshot diversification bolsters security and heightens reliability by diminishing the predictability of software. We therefore plan to investigate the feasibility of one-shot diversification in WebAssembly. However, we foresee several challenges, particularly the optimization of our previously presented tools.

In the context of WebAssembly, this process presents particularly challenging aspects since it is highly dynamic [17]. For instance, within a browser context, we need to ensure the WebAssembly binary varies not only per browser instance but also per tab process (webpage tab). In the backend, we must guarantee that the WebAssembly binary is unique per cold spawn [50, 63]. Hence, it becomes necessary to ensure that the diversification process is both rapid and efficient, capable of generating a vast number of variants within a brief timespan. Specifically, this entails producing millions of unique and diverse variants every second.

In addition to swift variant generation, a targeted diversification process is also necessary. For example, as shown in Chapter 4, feedback-guided diversification can quickly produce variants with specific objectives, such as evading malware. Therefore, while we diversify, we need to be able to discard those variants that offer fewer benefits based on custom feedback functions. For example, this process might require the inclusion of concepts like performance impact, variant's distribution, and variant's management. Performance impact, in particular, needs careful evaluation, given that WebAssembly is often used for applications sensitive to performance. Furthermore, distributing and managing diversified WebAssembly modules could become complex, especially due to the lack of a standard for WebAssembly module management or registry [12]. Besides, this complexity includes managing updates and ensuring compatibility across all diversified instances.

REFERENCES

[1] M. R. Cox, *Cinderella: Three hundred and forty-five variants of Cinderella, Catskin, and Cap o'Rushes*. No. 31, Folk-lore Society, 1893.

[2] Tim Berners-Lee, "The WorldWideWeb Browser." `https://www.w3.org/People/Berners-Lee/WorldWideWeb.html`, 1990.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The Essence of JavaScript," in *ECOOP 2010 - Object-Oriented Programming*, vol. 6183, pp. 126–150, 2010.

[4] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and Reliable Browser Identification With Javascript Engine Fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, p. 4, Citeseer, 2013.

[5] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript Instrumentation for Browser Security," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pp. 237–249, 2007.

[6] Y. Ko, T. Rezk, and M. Serrano, "SecureJS Compiler: Portable Memory Isolation in JavaScript," in *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing*, pp. 1265–1274, 2021.

[7] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the Web Up to Speed With WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pp. 185–200, 2017.

[8] C. Watt, "Mechanising and Verifying the WebAssembly Specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP*, pp. 53–65, 2018.

[9] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a Practical Path to Library Sandboxing," *CoRR*, vol. abs/1912.02285, 2019.

[10] P. Mendki, "Evaluating Webassembly Enabled Serverless Approach for Edge Computing," in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[11] M. Jacobsson and J. Willén, "Virtual Machine Execution for Wearables Based on WebAssembly," in *13th EAI International Conference on Body Area Networks, BODYNETS*, pp. 381–389, 2018.

[12] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "WebAssembly as a Common Layer for the Cloud-Edge Continuum," in *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, p. 3–8, 2022.

[13] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, "Exploring the Use of WebAssembly in HPC," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, p. 92–106, 2023.

[14] J. Cabrera-Arteaga, M. Monperrus, and B. Baudry, "Scalable Comparison of JavaScript V8 Bytecode Traces," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL at SPLASH 2019*, pp. 22–31, 2019.

[15] NSA, "National Cyber Leap Year." `https://www.nitrd.gov/nitrdgroups/index.php?title=National_Cyber_Leap_Year`, 2021.

[16] G. Goth, "Addressing the Monoculture," *IEEE Security & Privacy*, vol. 1, no. 06, pp. 8–10, 2003.

[17] M. N. Hoque and K. A. Harras, "WebAssembly for Edge Computing: Potential and Challenges," *IEEE Communications Standards Magazine*, vol. 6, no. 4, pp. 68–73, 2022.

[18] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port Contention Goes Portable: Port Contention Side Channels in Web Browsers," in *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security*, pp. 1182–1194, 2022.

[19] S. Song, S. Park, and D. Kwon, "metaSafer: A Technique to Detect Heap Metadata Corruption in WebAssembly," *IEEE Access*, vol. 11, pp. 124887–124898, 2023.

[20] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium*, pp. 217–234, 2020.

[21] Q. Stiévenart, C. D. Roover, and M. Ghafari, "Security Risks of Porting C Programs to Webassembly," in *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing*, pp. 1713–1722, 2022.

[22] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by Key-extraction Cache Attacks from Portable Code," *IACR Cryptol. ePrint Arch.*, p. 119, 2018.

[23] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pp. 2109–2122, 2018.

[24] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "Thieves in the Browser: Web-based Cryptojacking in the Wild," in *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, pp. 4:1–4:10, ACM, 2019.

[25] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selçuk, "In-browser Cryptomining for Good: An Untold Story," in *IEEE International Conference on Decentralized Applications and Infrastructures, DAPPS 2021, Online Event, August 23-26, 2021*, pp. 20–29, IEEE, 2021.

[26] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pp. 1714–1730, 2018.

[27] A. Romano, Y. Zheng, and W. Wang, "MinerRay: Semantics-aware Analysis for Ever-evolving Cryptojacking Detection," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 1129–1140, IEEE, 2020.

[28] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "MINOS: A Lightweight Real-time Cryptojacking Detection System," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021.

[29] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks," in *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS*, vol. 11099, pp. 122–142, 2018.

[30] J. D. P. Rodriguez and J. Posegga, "RAPID: Resource and API-based Detection Against In-browser Miners," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pp. 313–326, ACM, 2018.

[31] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting In-browser Covert Cryptocurrency Mining in the Wild," in *The World Wide Web Conference, WWW*, pp. 840–852, 2019.

[32] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of Cyber Moving Targets," *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166*, 2013.

[33] F. B. Cohen, "Operating System Protection Through Program Evolution.," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[34] S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pp. 67–72, 1997.

[35] M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the Internet virus of November 1988," in *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pp. 326–343, 1989.

[36] J. C. Arteaga, O. F. Malivitsis, O. L. V. Pérez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *Proceedings of MadWEB, NDSS*, 2021.

[37] J. Cabrera-Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-variant Execution at the Edge," in *Proceedings of the 9th ACM Workshop on Moving Target Defense, MTD*, pp. 11–22, ACM, 2022.

[38] J. Cabrera-Arteaga, N. FitzGerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *CoRR*, vol. abs/2309.07638, 2023.

[39] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "WebAssembly Diversification for Malware Evasion," *Comput. Secur.*, vol. 131, p. 103296, 2023.

[40] J. Cabrera Arteaga, "Artificial Software Diversification for WebAssembly," No. 2022:52 in TRITA-EECS-AVL, p. 112, 2022. `https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-317331`.

[41] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

[42] D. Bryant, "WebAssembly Outside the Browser: A New Foundation for Pervasive Computing," in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[43] B. Spies and M. Mock, "An Evaluation of WebAssembly in Non-web Environments," in *XLVII Latin American Computing Conference, CLEI 2021, Cartago, Costa Rica, October 25-29, 2021*, pp. 1–10, IEEE, 2021.

[44] E. Wen and G. Weber, "Wasmachine: Bring IoT up to Speed with A WebAssembly OS," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2020, Austin, TX, USA, March 23-27, 2020*, pp. 1–4, IEEE, 2020.

[45] A. Hilbig, D. Lehmann, and M. Pradel, "An Empirical Study of Real-world WebAssembly Binaries: Security, Languages, Use Cases," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pp. 2696–2708, 2021.

[46] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the Performance of Webassembly Applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, p. 533–549, 2021.

[47] L. Wagner, M. Mayer, A. Marino, A. S. Nezhad, H. Zwaan, and I. Malavolta, "On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE 2023, Oulu, Finland, June 14-16, 2023*, pp. 72–82, ACM, 2023.

[48] B. L. Titzer, "Whose baseline compiler is it anyway?," *arXiv e-prints*, p. arXiv:2305.13241, May 2023.

[49] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "WebAssembly Modules as Lightweight Containers for Liquid IoT Applications," in *Proceedings of Web Engineering - 21st International Conference, ICWE*, vol. 12706, pp. 328–336, 2021.

[50] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge," in *Middleware '20: 21st International Middleware Conference*, pp. 265–279, 2020.

[51] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, apr 2017.

[52] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "SecWasm: Information Flow Control for WebAssembly," in *Static Analysis - 29th International Symposium, SAS*, vol. 13790 of *Lecture Notes in Computer Science*, pp. 74–103, Springer, 2022.

[53] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for WebAssembly," *Comput. Secur.*, vol. 118, p. 102745, 2022.

[54] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic Execution for WebAssembly," Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[55] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 77:1–77:29, 2019.

[56] R. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," in *IEEE Secure Development Conference, SecDev 2021, Atlanta, GA, USA, October 18-20, 2021*, pp. 94–102, IEEE, 2021.

[57] Q. Stiévenart and C. De Roover, "Wassail: A WebAssembly Static Analysis Library," in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[58] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "WasmA: A Static WebAssembly Analysis Framework for Everyone," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, pp. 753–757, 2023.

[59] W. Fu, R. Lin, and D. Inge, "TaintAssembly: Taint-based Information Flow Control Tracking for WebAssembly," *CoRR*, vol. abs/1802.01050, 2018.

[60] Q. Stiévenart, D. Binkley, and C. De Roover, "Dynamic Slicing of WebAssembly Binaries," in *39th IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2023.

[61] Q. Stiévenart, D. W. Binkley, and C. D. Roover, "Static Stack-preserving Intra-procedural Slicing of WebAssembly Binaries," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 2031–2042, ACM, 2022.

[62] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pp. 1045–1058, 2019.

[63] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. M. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium, USENIX*, pp. 1433–1450, 2021.

[64] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation Without Taxation: Near-Zero-cost Transitions for WebAssembly And SFI," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–30, 2022.

[65] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "WaVe: A Verifiably Secure WebAssembly Sandboxing Runtime," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pp. 2940–2955, IEEE, 2023.

[66] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA*, vol. 11543, pp. 23–42, 2019.

[67] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A First Look at Code Obfuscation for WebAssembly," in *WiSec '22: 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 140–145, 2022.

[68] B. Baudry and M. Monperrus, "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 16:1–16:26, 2015.

[69] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.

[70] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing Performance vs. Accuracy Trade-offs With Loop Perforation," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, pp. 124–134, 2011.

[71] Avizienis and Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[72] T. Y. Chen, F. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, 2010.

[73] T. Jackson, *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine, 2012.

[74] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, jun 2014.

[75] G. R. Lundquist, V. Mohan, and K. W. Hamlen, "Searching for Software Diversity: Attaining Artificial Diversity through Program Synthesis," in *Proceedings of the 2016 New Security Paradigms Workshop*, NSPW '16, p. 80–91, 2016.

[76] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pp. 30–37, 1999.

[77] I. Gashi, P. Popov, and L. Strigini, "Fault Diversity among Off-The-Shelf SQL Database Servers," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, p. 389, 2004.

[78] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 96–109, 1986.

[79] B. Randell, "System Structure for Software Fault Tolerance," *SIGPLAN Not.*, vol. 10, p. 437–449, apr 1975.

[80] N. Harrand, *Software Diversity for Third-Party Dependencies*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2022.

[81] J. V. Cleemput, B. Coppens, and B. D. Sutter, "Compiler Mitigations for Time Attacks on Modern X86 Processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 23:1–23:20, 2012.

[82] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided Automated Software Diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pp. 23:1–23:11, IEEE Computer Society, 2013.

[83] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits," in *Proceedings of the USENIX Security Symposium*, 2003.

[84] S. Bhatkar and D. C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," in *Proceedings of the 14th USENIX*, 2005.

[85] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pp. 16–27, 1990.

[86] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-channel Attacks Through Dynamic Software Diversity," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[87] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pp. 1101–1116, may 2022.

[88] M. T. Aga and T. M. Austin, "Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO*, pp. 26–36, 2019.

[89] S. Lee, H. Kang, J. Jang, and B. B. Kang, "SaVioR: Thwarting Stack-based Memory Safety Violations by Randomizing Stack Layout," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 4, pp. 2559–2575, 2022.

[90] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended Protection against Stack Smashing Attacks without Performance Loss," in *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pp. 429–438, IEEE Computer Society, 2006.

[91] Y. Xu, Y. Solihin, and X. Shen, "MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pp. 987–1000, 2020.

[92] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-injection Attacks With Instruction-set Randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS*, pp. 272–280, 2003.

[93] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS*, pp. 281–289, 2003.

[94] M. Chew and D. Song, "Mitigating Buffer Overflows by Operating System Randomization," Tech. Rep. CS-02-197, Carnegie Mellon University, 2002.

[95] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J. Lanet, "Runtime Code Polymorphism as a Protection Against Side Channel Attacks," in *Proceedings of Information Security Theory and Practice - 10th IFIP WG 11.2 International Conference, WISTP*, vol. 9895, pp. 136–152, 2016.

[96] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," *CoRR*, vol. abs/2308.03123, 2023.

[97] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[98] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. Saw, and R. Venkatesan, "The Superdiversifier: Peephole Individualization for Software Protection,"

in *Proceedings of Advances in Information and Computer Security, Third International Workshop on Security, IWSEC 2008*, vol. 5312, pp. 100–120, 2008.

[99] M. Henry, "Superoptimizer: A Look at the Smallest Program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Nov 1987.

[100] V. Le, M. Afshari, and Z. Su, "Compiler Validation via Equivalence Modulo Inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 216–226, 2014.

[101] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic Program Alignment for Equivalence Checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 1027–1040, 2019.

[102] V. Le, C. Sun, and Z. Su, "Finding Deep Compiler Bugs via Guided Stochastic Program Mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, p. 386–399, 2015.

[103] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," vol. 15, p. 281–312, sep 2014.

[104] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants," ISSTA 2014, p. 149–159, 2014.

[105] M. Zalewski, "American Fuzzy Lop," 2017.

[106] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, "ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers," *CoRR*, vol. abs/2305.14591, 2023.

[107] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[108] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Counterexample Guided Inductive Synthesis Modulo Theories," in *Proceedings of Computer Aided Verification - 30th International Conference, CAV*, vol. 10981, pp. 270–288, 2018.

[109] P. M. Phothilimthana, A. Thakur, R. Bodík, and D. Dhurjati, "Scaling up Superoptimization," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pp. 297–310, 2016.

[110] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding Information in Program Binaries," in *Information and Communications Security, 6th International Conference, ICICS*, vol. 3269, pp. 187–199, 2004.

[111] V. Singhal, A. A. Pillai, C. Saumya, M. Kulkarni, and A. Machiry, "Cornucopia : A Framework for Feedback Guided Generation of Binaries," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 27:1–27:13, ACM, 2022.

[112] B. Cox and D. Evans, "N-Variant Systems: A Secretless Framework for Security through Diversity," in *Proceedings of the 15th USENIX*, 2006.

[113] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified Process replicæ for Defeating Memory Error Exploits," in *Proceedings of the 26th IEEE International Performance Computing and Communications Conference, IPCCC 2007, April 11-13, 2007, New Orleans, Louisiana, USA*, pp. 434–441, IEEE Computer Society, 2007.

[114] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-variant Program Execution on a Multicore Processor," tech. rep., Technical Report 07-13, School of Information and Computer Sciences, UCIrvine, 2007.

[115] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-oriented Programming," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[116] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 34, no. 8, pp. 1320–1333, 2015.

[117] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated Software Diversity," in *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, pp. 77–98, 2011.

[118] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can Code Polymorphism Limit Information Leakage?," in *Proceedings of Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP*, vol. 6633, pp. 1–21, 2011.

[119] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dMVX: Secure and Efficient Multi-variant Execution in a Distributed Setting," in *EuroSec '21: Proceedings of the 14th European Workshop on Systems Security, Virtual Event / Edinburgh, Scotland, UK, April 26, 2021*, pp. 41–47, ACM, 2021.

[120] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere, "Instruction Set Limitation in Support of Software Diversity," pp. 152–165, 2009.

[121] R. Tsoupidi, R. C. Lozano, and B. Baudry, "Constraint-based Diversification of JOP Gadgets," *J. Artif. Intell. Res.*, vol. 72, pp. 1471–1505, 2021.

[122] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pp. 313–324, 2014.

[123] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 661–678, Aug. 2017.

[124] H. Bostani and V. Moonsamy, "EvadeDroid: A Practical Evasion Attack on Machine Learning for Black-box Android Malware Detection," *CoRR*, vol. abs/2110.03301, 2021.

[125] D. D. Yao, X. Shu, L. Cheng, and S. J. Stolfo, *Anomaly Detection as a Service: Challenges, Advances, and Opportunities*. Synthesis Lectures on Information Security, Privacy, and Trust, Morgan & Claypool Publishers, 2017.

[126] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, 1998.

[127] Y. Fang, C. Huang, L. Liu, and M. Xue, "Research on Malicious JavaScript Detection Technology Based on LSTM," *IEEE Access*, vol. 6, pp. 59118–59125, 2018.

[128] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, ", : SFI safety for native-compiled Wasm," *Network and Distributed Systems Security (NDSS) Symposium*, 2021.

[129] F. Cohen, "Computer Viruses," in *Proceedings of the 7th DoD/NBS Computer Security Conference 1984*, pp. 240–263, 1986.

[130] R. L. Castro, C. Schmitt, and G. D. Rodosek, "ARMED: How Automatic Malware Modifications Can Evade Static Detection?," in *2019 5th International Conference on Information Management (ICIM)*, pp. 20–27, 2019.

[131] R. L. Castro, C. Schmitt, and G. Dreo, "AIMED: Evolving Malware with Genetic Programming to Evade Detection," in *18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 13th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, August 5-8, 2019*, pp. 240–247, IEEE, 2019.

[132] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster, "WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention," FSE 2016, p. 205–216, 2016.

[133] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020.

[134] M. W. J. Chua and V. Balachandran, "Effectiveness of Android Obfuscation on Evading Anti-malware," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY*, pp. 143–145, 2018.

[135] P. Dasgupta and Z. Osman, "A Comparison of State-of-the-art Techniques for Generating Adversarial Malware Binaries," *CoRR*, vol. abs/2111.11487, 2021.

[136] G. Lu and S. K. Debray, "Weaknesses in Defenses against Web-borne Malware," in *Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013*, vol. 7967, pp. 139–149, Springer, 2013.

[137] M. Payer, "Embracing the New Threat: Towards Automatically Self-diversifying Malware," in *Proceedings of The Symposium on Security for Asia Network*, pp. 1–5, 2014.

[138] N. Loose, F. Mächtle, C. Pott, V. Bezsmertnyi, and T. Eisenbarth, "Madvex: Instrumentation-based Adversarial Attacks on Machine Learning Malware Detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023*, vol. 13959 of *Lecture Notes in Computer Science*, pp. 69–88, 2023.

[139] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ch. 1, pp. 28–31. 1986.

[140] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *CoRR*, vol. abs/1711.04422, 2017.

[141] B. G. Ryder, "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[142] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. M. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pp. 1433–1450, 2021.

[143] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and Extensible Equality Saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021.

[144] "Stop a wasm compiler bug before it becomes a problem | fastly." `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`, 2021.

[145] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "babble: Learning Better Abstractions with E-Graphs and Anti-unification," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 396–424, 2023.

[146] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality Saturation: A New Approach to Optimization," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pp. 264–276, 2009.

[147] T. D. Morgan and J. W. Morgan, "Web Timing Attacks Made Practical," *Black Hat*, 2015.

[148] T. Schnitzler, K. Kohls, E. Bitsikas, and C. Pöpper, "Hope of Delivery: Extracting User Locations From Mobile Instant Messengers," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023.

[149] Mozilla, "Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta ," 2019.

[150] F. Cohen, "Computer Viruses: Theory and Experiments," *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, 1987.

[151] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *meltdownattack.com*, 2018.

[152] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-resolution Microarchitectural Attacks in JavaScript," in *Financial Cryptography and Data Security - 21st International Conference, FC*, vol. 10322, pp. 247–267, 2017.

[153] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary Rewriting Without Control Flow Recovery," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, pp. 151–163, 2020.

[154] A. Nicholson, Q. Stiévenart, A. Mazidi, and M. Ghafari, "Wasmizer: Curating WebAssembly-driven Projects on GitHub," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 130–141, 2023.

[155] T. Y. Zhuo, Z. Yang, Z. Sun, Y. Wang, L. Li, X. Du, Z. Xing, and D. Lo, "Source Code Data Augmentation for Deep Learning: A Survey," *arXiv e-prints*, p. arXiv:2305.19915, May 2023.

[156] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U. O'Reilly, "Generating Adversarial Computer Programs using Optimized Obfuscations," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021.

[157] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 92:1–92:13, ACM, 2022.

[158] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging Machine Learning-based Clone Detectors via Semantic-preserving Code Transformations," *IEEE Trans. Software Eng.*, vol. 49, no. 5, pp. 3052–3070, 2023.

[159] H. Li, X. Zhou, L. A. Tuan, and C. Miao, "Rethinking Negative Pairs in Code Search," *arXiv preprint arXiv:2310.08069*, 2023.

[160] J. D. Seideman, *Transformation and Abstraction to Aid Comparison of Binary Executables Across Compilation Environments.* PhD thesis, City University of New York, 2023.

[161] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection," *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[162] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *2012 IEEE Symposium on Security and Privacy*, pp. 48–62, 2012.

[163] H. Jang, K. Yang, G. Lee, Y. Na, J. D. Seideman, S. Luo, H. Lee, and S. Dietrich, "QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection," in *ICT Systems Security and Privacy Protection*, pp. 66–82, 2021.

Part II

Included papers

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

https://www.sciencedirect.com/science/article/pii/S0167404823002067

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Submitted to Computers & Security, under revision

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), Workshop on Measurements, Attacks, and Defenses for the Web

https://doi.org/10.14722/madweb.2021.23004

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Workshop on Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228